

Visual C# .NET

Copyright:
This Edition:
Author:

Home and Learn
December 2012
Ken Carney

All rights reserved

Visual C# .NET – Contents

| | |
|---|------------|
| INTRODUCTION | 7 |
| WHAT YOU NEED TO DO THE COURSE | 7 |
| THE FREE VISUAL C# EXPRESS EDITION | 7 |
| ADDITIONAL FILES | 8 |
| GETTING STARTED WITH C# | 9 |
| A SIMPLE C# CONSOLE APPLICATION..... | 11 |
| SAVING YOUR WORK | 17 |
| YOUR FIRST LINE OF CODE..... | 21 |
| RUNNING YOUR PROGRAMMES | 25 |
| YOUR WINDOWS FIRST FORM..... | 28 |
| ADDING CONTROLS TO A BLANK FORM | 37 |
| PROPERTIES OF A CONTROL | 38 |
| ADDING CODE TO A BUTTON..... | 43 |
| A MESSAGEBOX | 44 |
| OTHER THINGS TO DO WITH THE MESSAGE BOX | 47 |
| OTHER BUTTON OPTIONS | 49 |
| ADDING ICONS TO A MESSAGE BOX..... | 50 |
| VARIABLES - STRINGS | 52 |
| STRING VARIABLES | 52 |
| ASSIGNING TEXT TO A STRING VARIABLE | 61 |
| CONCATENATION | 61 |
| USING OTHER CONTROLS TO DISPLAY TEXT..... | 63 |
| VARIABLES - NUMBERS..... | 67 |
| INTEGERS | 67 |
| DOUBLES AND FLOAT | 71 |
| DOUBLE VALUES..... | 74 |
| SIMPLE ADDITION | 76 |
| ADDING UP WITH FLOAT VARIABLES..... | 80 |
| SUBTRACTION..... | 85 |
| MIXING SUBTRACTION AND ADDITION | 87 |
| OPERATOR PRECEDENCE | 88 |
| MULTIPLICATION AND DIVISION..... | 90 |
| GETTING NUMBERS FROM TEXT BOXES..... | 91 |
| CALCULATOR PROJECT..... | 98 |
| CODE THE NUMBER BUTTONS..... | 101 |
| THE PLUS BUTTON | 104 |
| THE EQUALS BUTTON..... | 105 |
| CONDITIONAL LOGIC | 108 |
| IF STATEMENTS | 108 |
| ELSE | 110 |
| ELSE ... IF | 111 |
| SWITCH STATEMENTS..... | 115 |
| CONDITIONAL OPERATORS | 118 |

| | |
|---|------------|
| LOOPING THE LOOP | 124 |
| FOR LOOPS | 124 |
| LOOP START VALUES AND LOOP END VALUES | 129 |
| TIMES TABLE PROGRAMME..... | 131 |
| DO LOOPS AND WHILE LOOPS | 138 |
| TRY PARSE..... | 139 |
| ADDING MENUS TO WINDOWS FORMS IN C#..... | 143 |
| HOW TO ADD A SUB MENU..... | 148 |
| MENU SHORTCUTS | 149 |
| ADDING CODE FOR YOUR MENU ITEMS | 154 |
| THE CODE TO VIEW AND INSERT IMAGES IN C# | 166 |
| THE OPEN FILE DIALOGUE BOX | 174 |
| THE SAVE MENU | 176 |
| CHECKBOXES AND RADIO BUTTONS..... | 179 |
| ERROR HANDLING AND DEBUGGING | 186 |
| ERRORS AT DESIGN-TIME | 186 |
| RUN-TIME ERRORS..... | 188 |
| LOGIC ERRORS..... | 190 |
| BREAKPOINTS..... | 191 |
| THE LOCALS WINDOW..... | 194 |
| TRY ... CATCH..... | 195 |
| METHODS | 199 |
| CALLING YOUR METHODS..... | 200 |
| PASSING VALUES TO YOUR METHODS..... | 201 |
| GETTING VALUES BACK FROM METHODS..... | 204 |
| UNDERSTANDING ARRAYS | 209 |
| WHAT IS AN ARRAY?..... | 209 |
| HOW TO SET UP AN ARRAY | 210 |
| ASSIGNING VALUES TO YOUR ARRAYS | 211 |
| ARRAYS AND LOOPS..... | 212 |
| USE A LOOP TO ASSIGN VALUES TO AN ARRAY | 215 |
| SET THE SIZE OF AN ARRAY AT RUNTIME | 217 |
| MULTI DIMENSIONAL ARRAYS..... | 220 |
| ARRAYS AND TEXT | 222 |
| THE FOREACH LOOP | 223 |
| COLLECTIONS..... | 224 |
| LISTS | 225 |
| ADDING ITEMS TO A LIST | 228 |
| SORTING A LIST..... | 229 |
| REMOVING ITEMS FROM A LIST..... | 230 |
| HASHTABLES | 231 |
| ENUMERATIONS..... | 233 |
| STRING MANIPULATION IN C#..... | 236 |
| STRING VARIABLES | 236 |
| TRIMMING UNWANTED CHARACTERS | 239 |
| THE CONTAINS METHOD | 241 |
| THE INDEXOF METHOD | 242 |
| THE INSERT METHOD | 245 |
| THE PADLEFT AND PADRIGHT METHODS..... | 245 |
| THE REMOVE METHOD..... | 246 |

| | |
|--|------------|
| THE REPLACE METHOD..... | 247 |
| THE SUBSTRING METHOD | 247 |
| THE SPLIT METHOD | 248 |
| THE JOIN METHOD | 249 |
| A C# .NET HANGMAN GAME..... | 250 |
| LINQ..... | 253 |
| LINQ METHODS AND PROPERTIES | 253 |
| THE SUM METHOD..... | 254 |
| MINIMUM AND MAXIMUM | 255 |
| AVERAGE | 255 |
| CONTAINS | 256 |
| ELEMENT AT | 256 |
| FIRST AND LAST | 257 |
| DISTINCT | 258 |
| TAKE AND INTERSECT | 259 |
| LINQ QUERIES..... | 262 |
| FROM, IN, SELECT | 262 |
| ORDER BY | 264 |
| WHERE | 265 |
| EVENTS | 267 |
| THE CLICK EVENT FOR BUTTONS | 267 |
| THE MOUSEDOWN EVENT | 268 |
| THE KEYDOWN EVENT | 270 |
| THE LEAVE EVENT | 271 |
| LISTBOX AND COMBOBOX EVENTS..... | 273 |
| CLASSES AND OBJECTS IN C# .NET | 277 |
| WHAT IS A CLASS? | 277 |
| WHAT IS AN OBJECT?..... | 277 |
| A HAPPY BIRTHDAY CLASS..... | 277 |
| CREATING OBJECTS FROM YOUR CLASSES..... | 280 |
| PASSING VALUES TO YOUR CLASSES..... | 282 |
| ADDING PROPERTIES TO YOUR CLASS | 283 |
| USING YOUR PROPERTIES | 286 |
| CLASS CONSTRUCTORS | 289 |
| INHERITANCE..... | 293 |
| METHOD OVERLOADING..... | 296 |
| STATIC METHODS..... | 298 |
| MANIPULATING FILES WITH C# .NET | 300 |
| HOW TO OPEN A TEXT FILE IN C#..... | 300 |
| READ A FILE LINE BY LINE | 304 |
| WRITE TO A TEXT FILE | 305 |
| APPENDING TEXT TO A FILE | 307 |
| COPY A FILE | 307 |
| MOVE A FILE..... | 308 |
| DELETE A FILE..... | 309 |
| SQL SERVER EXPRESS AND VISUAL C# .NET | 310 |
| WHAT IS SQL SERVER EXPRESS?..... | 310 |
| HOW TO CREATE A COMPACT SQL SERVER DATABASE..... | 311 |
| HOW TO CREATE TABLES IN YOUR SQL SERVER DATABASE | 315 |
| ADDING DATA TO A SQL SERVER DATABASE TABLE..... | 318 |
| CREATE A DATABASE PROJECT..... | 319 |

| | |
|--|------------|
| CONNECTING TO A SQL SERVER EXPRESS DATABASE | 320 |
| DATASETS AND DATA ADAPTERS..... | 322 |
| ACCESSING DATA FROM THE DATASET..... | 323 |
| DATABASE NAVIGATION BUTTONS | 326 |
| MOVE BACKWARDS THROUGH THE DATABASE | 328 |
| JUMP TO THE LAST RECORD IN YOUR DATABASE | 329 |
| JUMP TO THE FIRST RECORD IN YOUR DATABASE | 330 |
| ADD A NEW RECORD | 330 |
| UPDATE A RECORD | 334 |
| DELETE A RECORD | 335 |
| FINDING RECORDS..... | 336 |
| CREATING MULTIPLE FORMS..... | 341 |
| MODAL FORMS | 343 |
| GETTING AT THE VALUES ON OTHER FORMS..... | 345 |
| WORKING WITH DATES AND TIME IN VISUAL C# .NET | 351 |
| GRAPHICS | 356 |
| THE RECTANGLE CLASS | 357 |
| BRUSHES..... | 359 |
| DRAWING POLYGONS..... | 360 |
| DRAWING TEXT | 361 |
| HOW TO CREATE A CLASS LIBRARY FILE | 363 |
| C# CHARTS | 372 |
| ADDING A CHART TO A WINDOWS FORM..... | 372 |
| ADDING THE DATA FOR A CHART | 375 |
| ADDING DATA TO A CHART AT RUNTIME..... | 378 |
| CHART FORMATTING..... | 382 |
| CHART TITLES..... | 386 |
| FORMATTING CHART SERIES | 389 |
| FORMATTING THE CHARTAREA..... | 393 |
| LINE CHARTS..... | 397 |
| ADDING MARKERS..... | 406 |
| PIE CHARTS..... | 413 |
| ADDING DATA FOR A PIE CHART | 415 |
| SERIES CUSTOM COLOURS..... | 418 |
| EXPLODING PIE SLICES..... | 422 |
| 3D CHARTS..... | 424 |
| CREATE YOUR OWN BROWSER..... | 428 |
| THE WEBBROWSER CONTROL..... | 430 |
| NAVIGATION BUTTONS..... | 432 |
| GRAPHIC BUTTONS..... | 434 |
| ADDING TOOLTIPS..... | 438 |
| PICTURE VIEWER – A PROJECT..... | 440 |
| ADDING THE CONTROLS TO THE FORM | 440 |
| SELECTING IMAGES | 443 |
| ADDING IMAGES TO AN IMAGES LIST..... | 446 |
| ADDING IMAGES AND FILE PATHS TO THE LISTVIEW | 446 |
| THE BIGGER PICTURE | 449 |
| ZOOMING IN AND OUT..... | 451 |
| IMAGE EDITING | 454 |
| IMAGE ROTATION..... | 454 |

| | |
|----------------------------------|------------|
| IMAGE FLIPPING..... | 456 |
| CLONE AN IMAGE | 457 |
| MANIPULATING IMAGE PIXELS | 459 |
| INVERT PIXEL COLOURS | 465 |
| SAVING AN IMAGE..... | 466 |
| ANSWERS TO EXERCISES..... | 469 |

Introduction

Hello, and a very warm welcome to the Home and Learn computer book for C# .NET Programming (all versions, up to and including 2012). The software you need is set out below. We assume that you have absolutely no knowledge of programming. Throughout the course of this book you will learn the fundamentals of NET programming with the free edition of the Visual C# .NET software. And, of course, you will start writing your own programmes. By the end of the book, you will have acquired a good understanding of what programming is all about, and have the ability to take it further, if you so wish. At the very least, you will have given your brain a good work out!

We hope you enjoy your computer book, and our time together.

Before you make a start, though, please read the following brief sections.

What you need to do the course

To do this course you need the following:

- A PC running the Windows XP, Vista, Windows 7, or Windows 8 operating system
- The free Visual Studio Express Edition (any version up to and including 2010)
- An internet connection to download the extra files mentioned in this book

The Free Visual C# Express Edition

This book uses the free edition of Visual Studio. At the time of writing, you can download it from here:

<http://www.microsoft.com/visualstudio/eng/products/visual-studio-express-products>

If you have Windows 7 or 8 then you can get the version for Express Products 2012, **Express for Windows Desktop**. If you don't have Windows 7 or 8 then you may have problems installing version 2012. In which case, click the link for **2010 Express products**. Whichever version you get, though, our course will cover it.

We cannot, however, accept questions about the installation of the Microsoft software.

Additional Files

Throughout this book, you will see references to additional files. These can now be downloaded from our web site. Connect to the internet and go to the following web page:

www.homeandlearn.co.uk/downloads/downloads.html

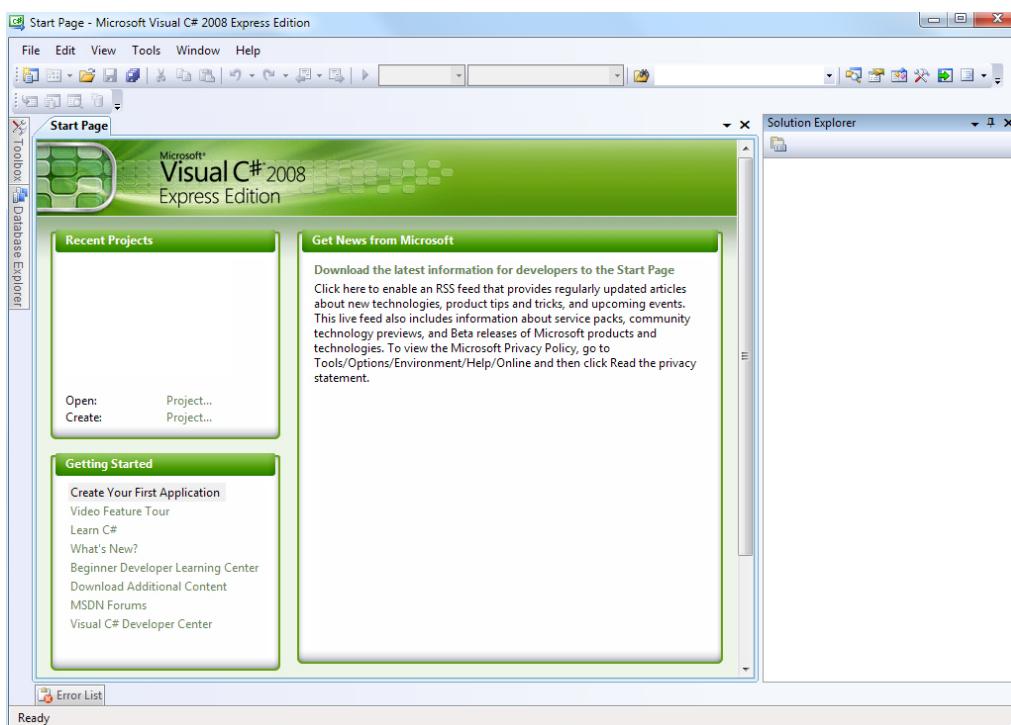
Once on the page, click the link for your course book, and save the Zip file to your own hard drive. If you have any problems downloading the files, please contact us at the following email address:

enquiry@homeandlearn.co.uk

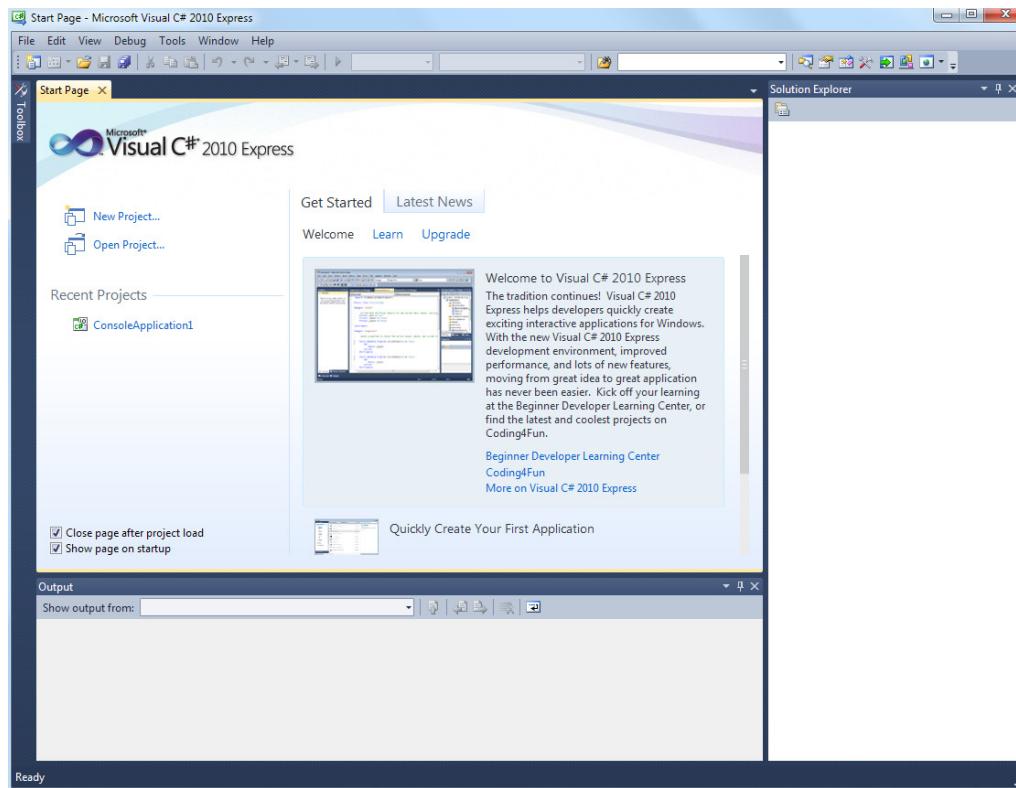
You can now make a start. Good luck with your programming!

Getting Started with C#

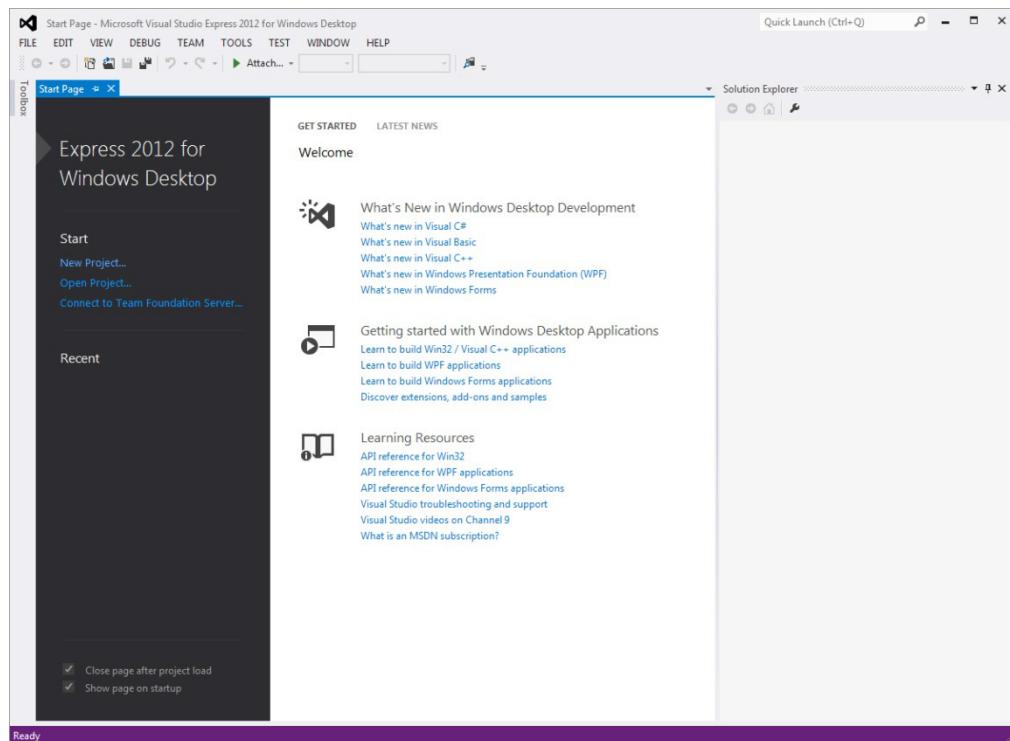
Open the Visual C# Express software from your programs menu. When you first open C# (pronounced C Sharp), you should see a screen something like this in Visual C# 2008:



If you have the 2010 version of Visual C# Express then your screen looks like this on the next page:



The Visual Studio 2012 opening screen will look like this:



When you're looking at this piece of software for the first time, it can seem hugely complex and daunting. The temptation is to think you're never going to get to grips with something so difficult. But don't worry - after a few lessons things will start to feel familiar, and you won't feel nearly half as intimidated as you do now!

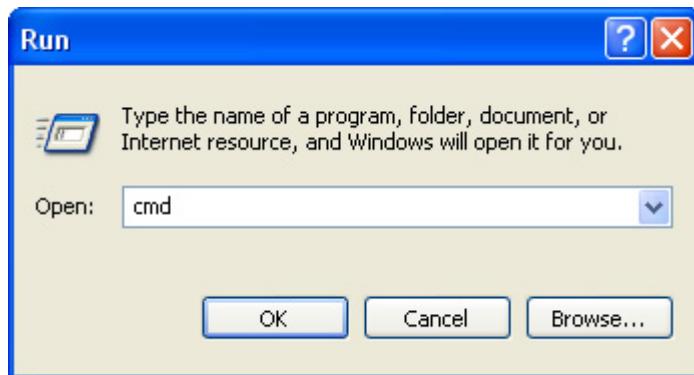
What we're going to do first is to create a very simple programme, so that you can see what makes up a C# project. By the end of this chapter, you'll have learned the following:

- How to create new projects
- What the Solution Explorer is
- The various files that make up of a C# project
- How to save your work
- How to run programmes
- The importance of the **Main** statement

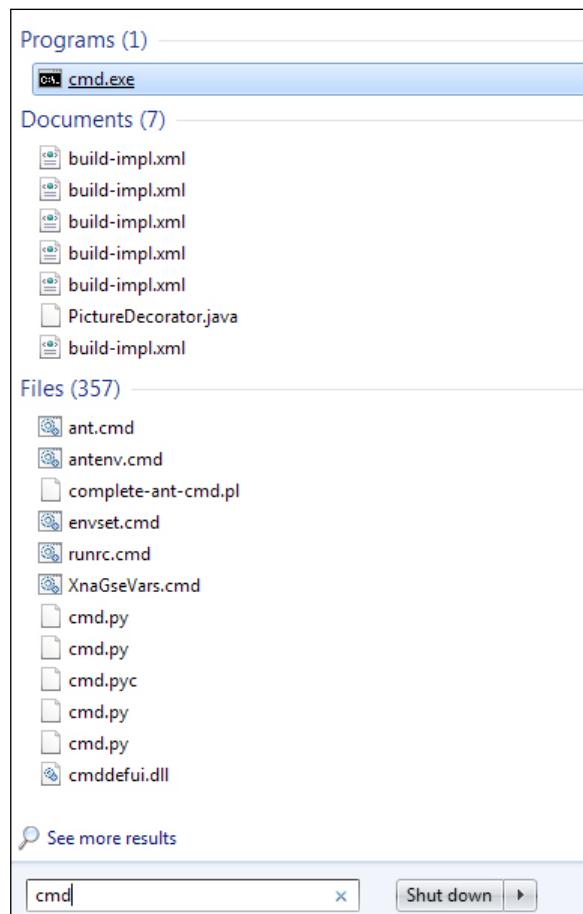
The simple programme we'll create is called a Console Application. We won't be doing much else with this type of application, as this is a book about Windows Forms Applications. Off we go then!

A Simple C# Console Application

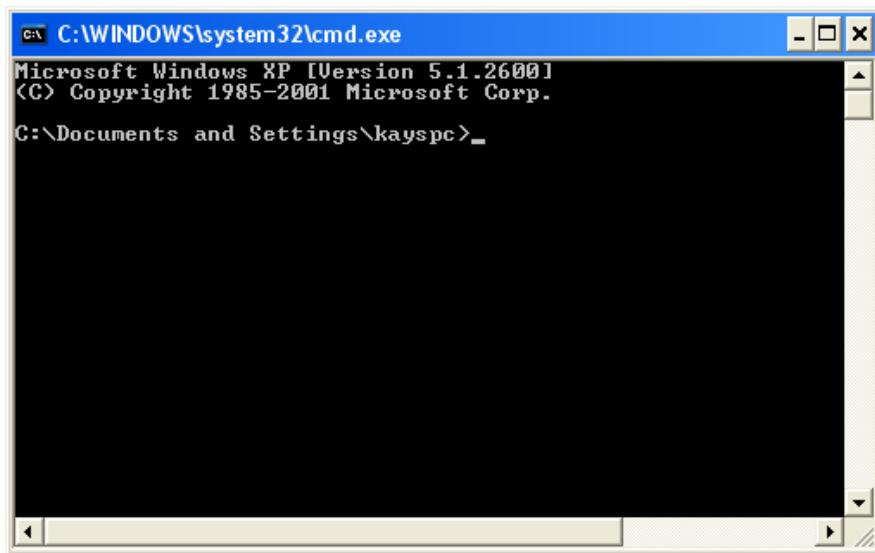
A Console Application is one that looks like a DOS window. If you don't know what these are, click your **Start** menu in the bottom left of your screen. In Windows XP, click on **Run**. From the dialogue box that appears, type **cmd**:



In Vista and Windows 7, type **cmd** in the search box at the bottom of the start menu. In Windows 8, the search box is on the Start Screen page. You'll then see the search results appear:

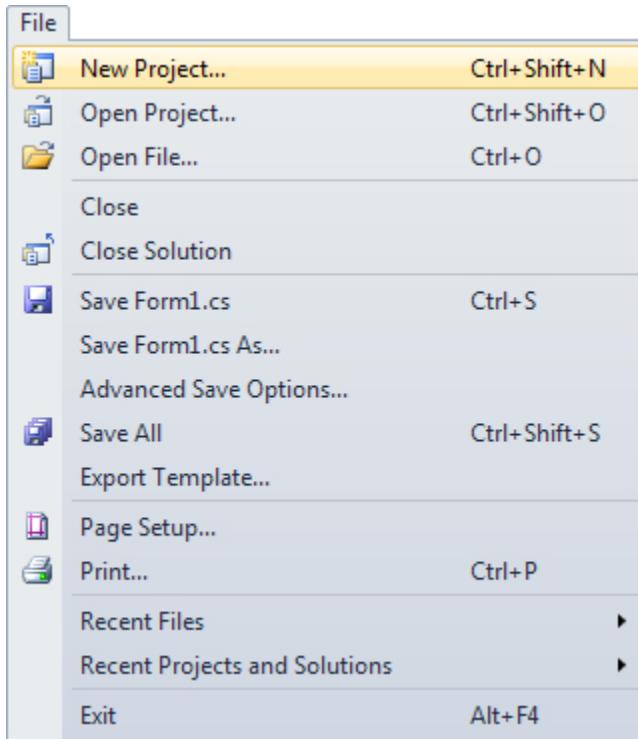


Click **cmd.exe** to see the console appear. Click OK and you'll see a black screen, like this one:

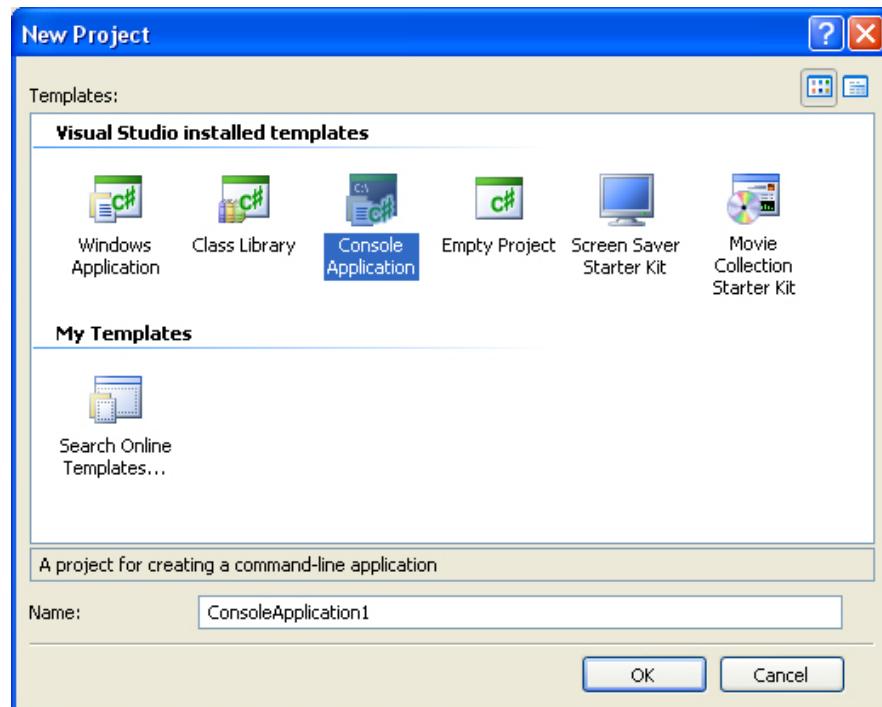


This is the type of window you'll see for our Console Application. When you create your Window forms, there's a whole lot of code to get used to. But Console Applications start off fairly simple, and you can see which part of the programme is the most important.

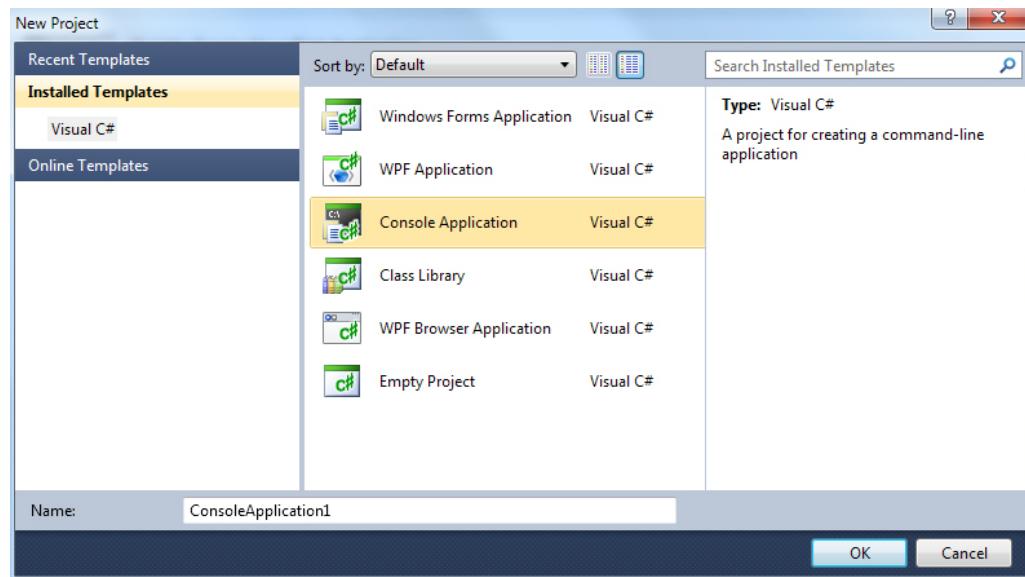
So with Visual C# Express open, click **File** from the menu bar at the top. From the **File** menu, select **New Project** (or click the New Project link on the left of the opening screen in versions 2010 and 2012):



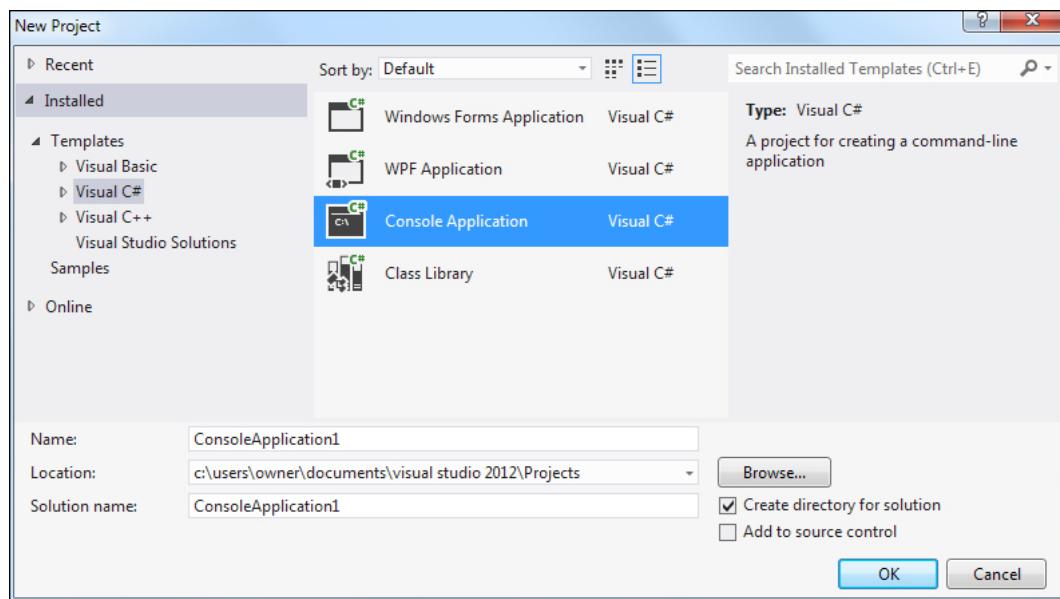
When you click on New Project, you'll see the following dialogue box appear in version 2008:



Or this one in version 2010 of the software:

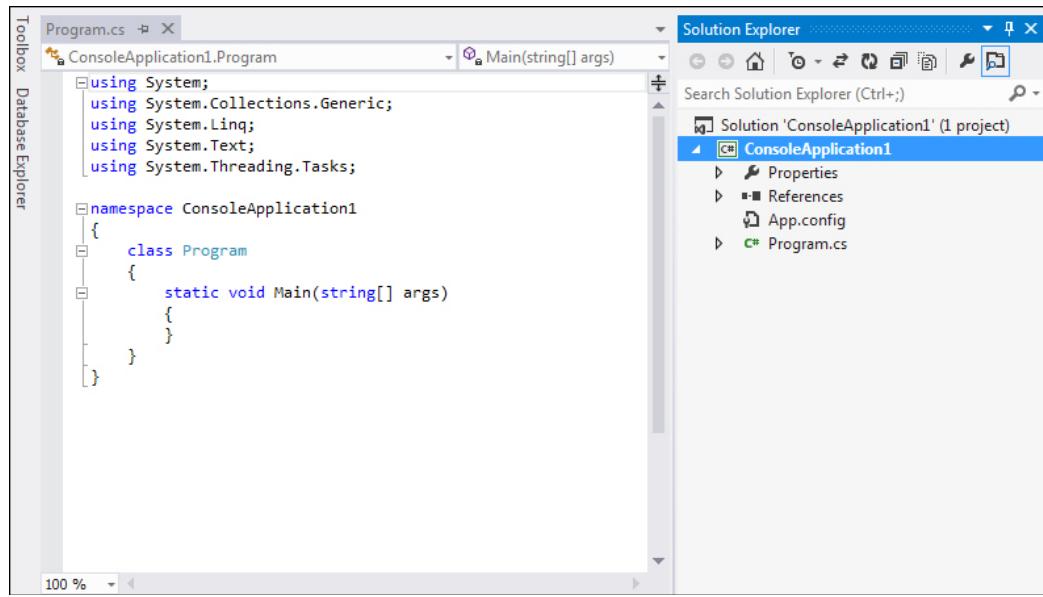


For 2012 users, click on **Templates** from the list on the left. Under Templates, click on **Visual C#**. You'll then see **Console Application** appear in the middle:



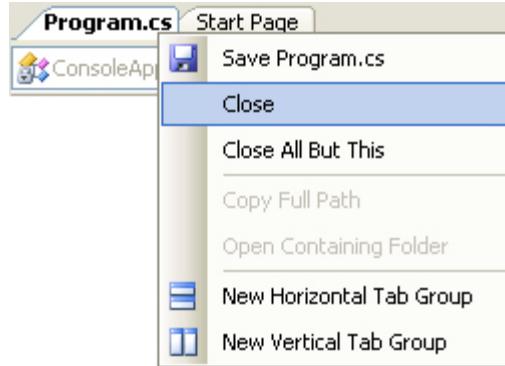
For all versions, the New Project dialogue box is where you select the type of project you want to create. If you only have the Express edition of Visual C#, the options are limited. For the rest of this book, we'll be creating Windows Applications. For now, select Console Application. Then click OK.

When you click OK, a new Console Application project will be created for you. Some code should be displayed (older versions will have fewer **using** statements at the top):

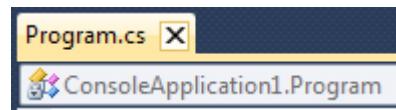


As well as the code, have a look on the right hand side and you'll see the Solution Explorer. This is where all the files for your project are.

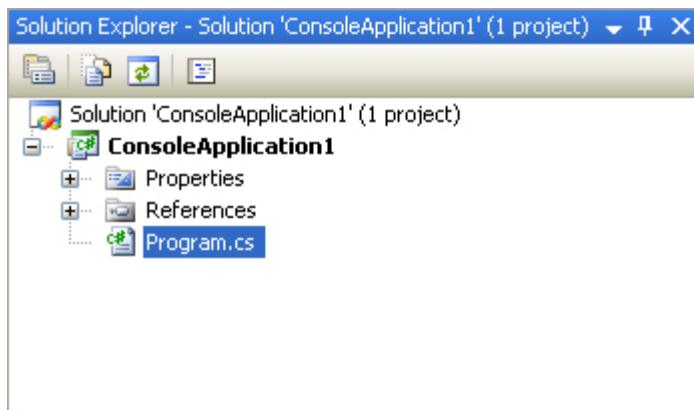
The code itself will look very complicated, if you're new to programming. We'll get to it shortly. For now, right click the **Program.cs** tab at the top, and click **Close** from the menu that appears:



Or just click the X in C# 2010 and 2012 :

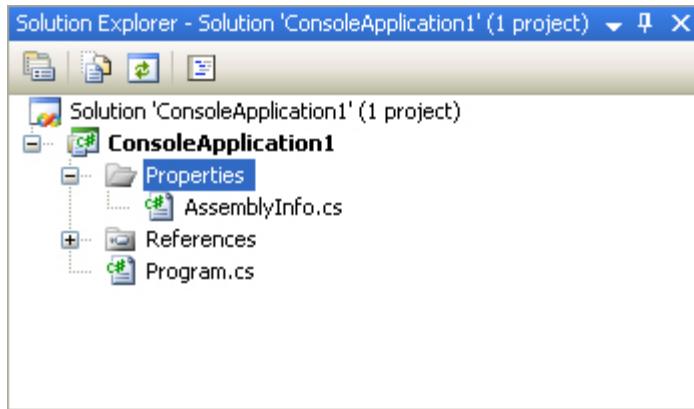


Now double click the **Program.cs** file in the Solution Explorer:



When you double click Program.cs, you should see the code reappear. So this code is the programme that will run when anyone starts your application.

Now click the plus symbol (arrow symbol in version 2012) next to **Properties** in the Solution Explorer above. You'll see the following:



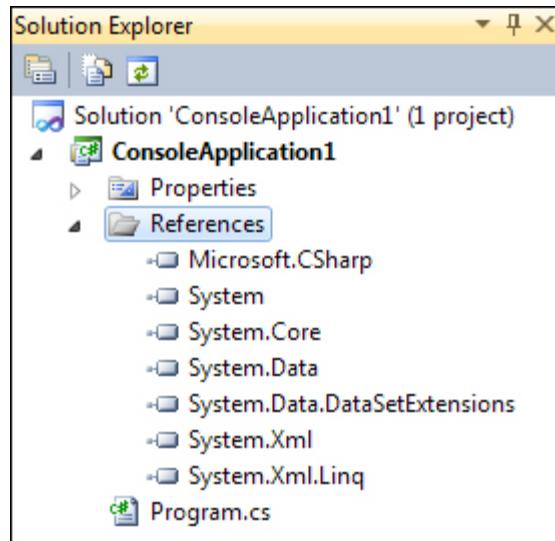
The file called AssemblyInfo.cs contains information about your programme. Double click this file to open it up and see the code. Here's just some of it:

```
[assembly: AssemblyTitle("ConsoleApplication1")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("ConsoleApplication1")]
[assembly: AssemblyCopyright("Copyright ©  2010")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

The reddish colour text is something you can change. You can add a Title, Description, Copyright, Trademark, etc.

But right click the AssemblyInfo.cs tab at the top, and click **Close** from the menu.

Now, in the Solution Explorer, click the plus symbol next to References:

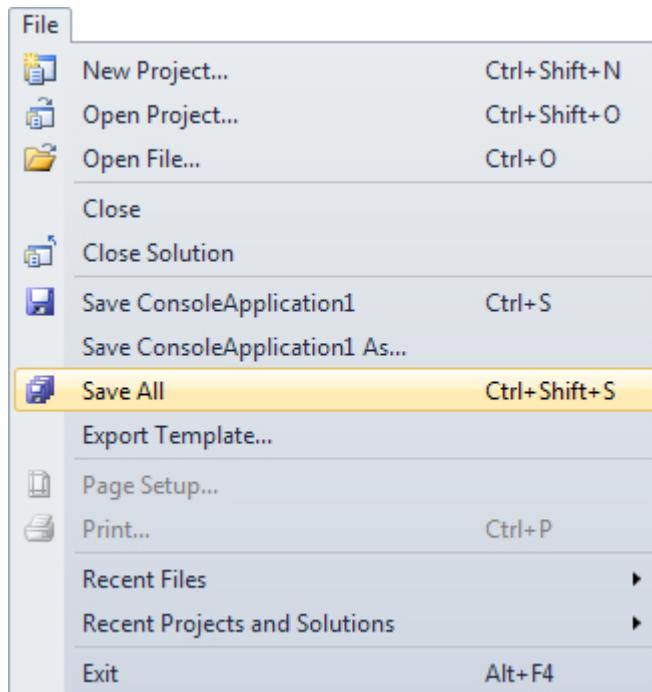


These are references to code built in to C# (you won't see as many entries in earlier versions of the software). Much later, you'll see how to add your own files to this section.

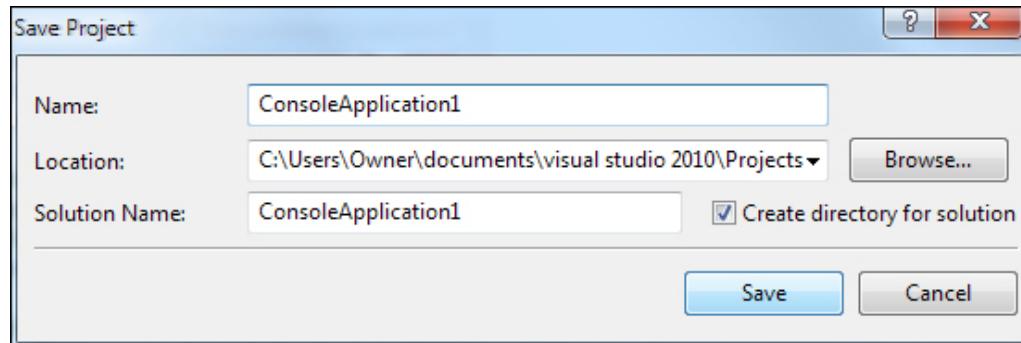
Before we add some code, let's save the project.

Saving your work

When you save your work, C# will create quite a few folders and files for you. Click **File** from the menu bar at the top, then Save All:



When you click Save All, you'll see the following dialogue box appear in versions 2008 and 2010:



(2012 users won't need to do anything here, as this above information was displayed when you created a new project earlier.)

You can type any name you like for your project. The default Name is **ConsoleApplication1**. Have a look at the location of the project, though:

C:\Users\Owner\documents\visual studio 2010\Projects

In XP, however, you'll see something like this:

C:\Documents and Settings\kayspc\My Documents\Visual Studio 2010\Projects

So it's going to be saved to the "documents" folder of this computer. In the "documents" folder you'll find another one called Visual Studio, followed by a year. In this folder there will be one called Projects.

Before clicking the Save button, make sure there is a tick in the box for "Create directory for solution". Then click **Save**.

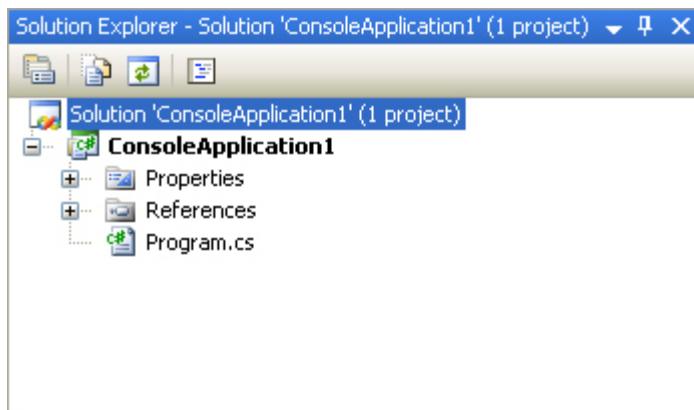
Now open up Windows Explorer (Hold down the Windows key on your keyboard, then press the letter "e"). Navigate to the folder location above. In the image below, we've used Windows Explorer to navigate to the Visual Studio 2010 folder:

| Documents library | | |
|--------------------|-------------|--|
| Visual Studio 2010 | | |
| Name | Type | Folder path |
| Backup Files | File folder | C:\Users\Owner\My Documents\Visual Studio 2010 |
| Code Snippets | File folder | C:\Users\Owner\My Documents\Visual Studio 2010 |
| Projects | File folder | C:\Users\Owner\My Documents\Visual Studio 2010 |
| Settings | File folder | C:\Users\Owner\My Documents\Visual Studio 2010 |
| StartPages | File folder | C:\Users\Owner\My Documents\Visual Studio 2010 |
| Templates | File folder | C:\Users\Owner\My Documents\Visual Studio 2010 |
| Visualizers | File folder | C:\Users\Owner\My Documents\Visual Studio 2010 |

Double click the **Projects** folder to see inside of it. You should see a folder called **ConsoleApplication1**. Double click this folder and you'll see the following:

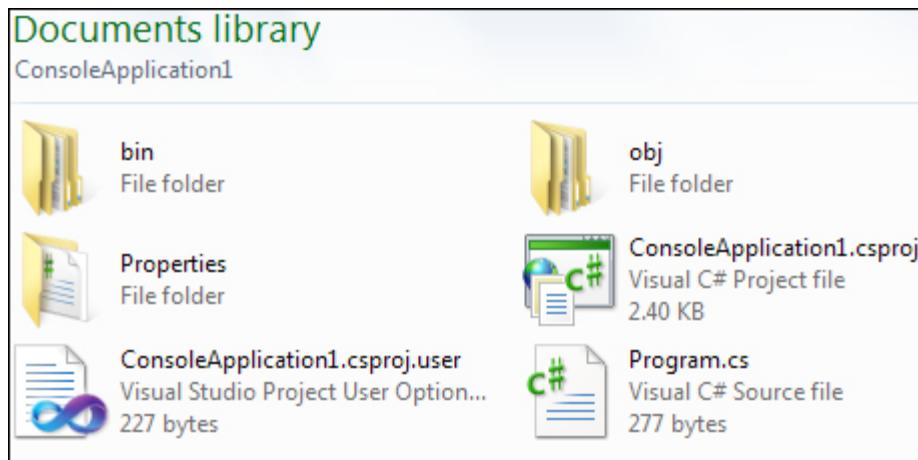


So there's another folder called **ConsoleApplication1**. There are also two files: one that ends in **sln**, and one that ends in **suo**. The **sln** file is the entire solution. Have a look at the Solution Explorer again:

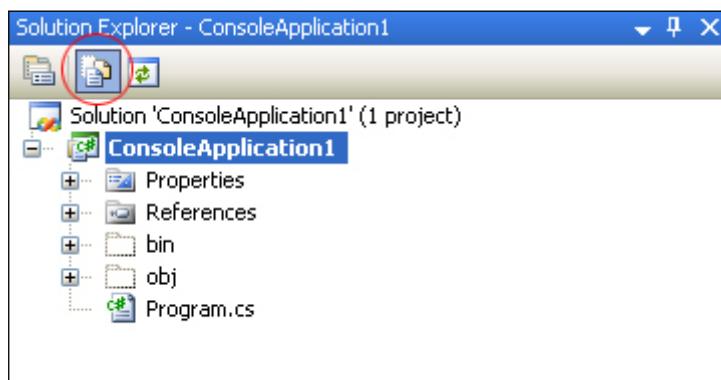


The one highlighted in blue at the top refers to the **sln** file. The **suo** file contains information about the Visual Studio environment – whether the plus symbols are expanded in the Solution Explorer, what other files you have open in this project, and a whole host of other settings. (If you can't see the **suo** file click **Tools > Folder Option** in Windows Explorer. In Vista and Windows 7/8, you may have to click **Organise > Layout > Menu Bar** first. Click the **View** tab, and select the option for “Show hidden files and folders”.)

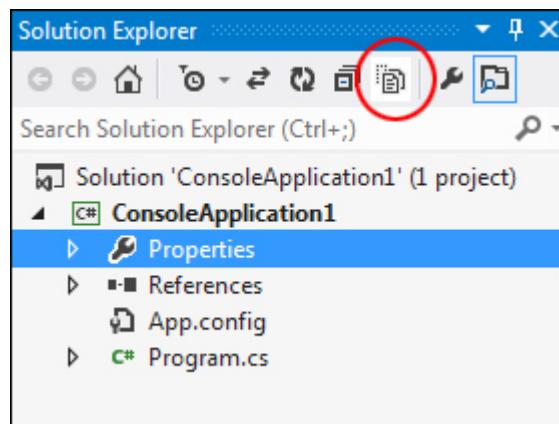
Double click your **ConsoleApplication1** folder, though, to see inside of it:



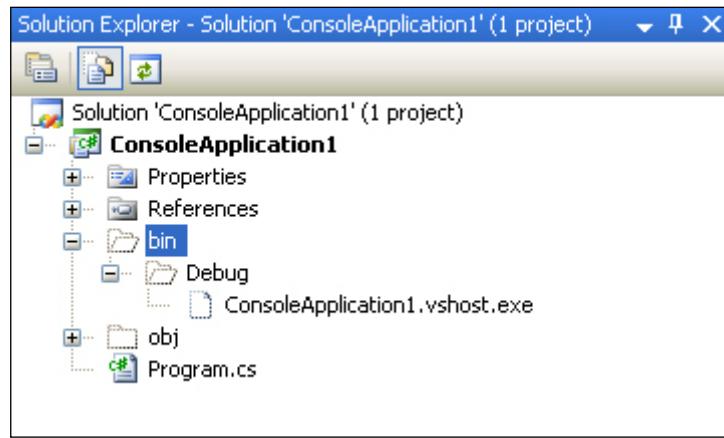
Now we have three more folders and three files (two files in earlier versions of c#). You can see the **bin** and **obj** folders in the Solution Explorer:



Click **ConsoleApplication1**, second from the top. Then click the icon for **Show all Files**, circled in red in the image above. To see All Files in version 2012, click the symbol circled in the image below:



The **bin** and **obj** folders will appear. Click the plus symbol or arrows to see what's inside of these folders:



The important one for us is the **Debug** folder under **bin** (there'll be an extra file ending in **.manifest** in c# 2010). You'll see why it's important in a moment. However, it's time to write some code!

Your first line of code

The only thing we'll do with the code is to write some text to the screen. But here's the code that Visual C# prepares for you when you first create a Console Application:

```
Program.cs  X
ConsoleApplication1.Program
Main(string[] args)

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

A screenshot of the code editor window showing the "Program.cs" file. The code defines a namespace "ConsoleApplication1" and a class "Program" with a static void Main method. The code is syntax-highlighted with blue for keywords like "using", "namespace", and "static". Braces and punctuation are in black.

For now, ignore the lines that start with **using** as we'll get to them later in the book. (The image above is from version 2012 - earlier versions will have fewer **using** lines). But they add references to in-built code. The **namespace** line includes the name of your application. A namespace is a way to group related code together. Again, don't worry about the term **namespace**, as you'll learn about this later.

The thing that's important above is the word **class**. All your code will be written in classes. This one is called **Program** (you can call them anything you like, as long as C# hasn't taken the word for itself). But think of a **class** as a segment of code that you give a name to.

Inside of the class called Program there is this code:

```
static void Main(string[] args)
{
}
```

This piece of code is something called a Method. The name of the Method above is **Main**. When you run your programme, C# looks for a Method called Main. It uses the Main Method as the starting point for your programmes. It then executes any code between those two curly brackets. The blue words above are all special words – keywords. You'll learn more about them in later chapters.

But position your cursor after the first curly bracket, and then hit the enter key on your keyboard:

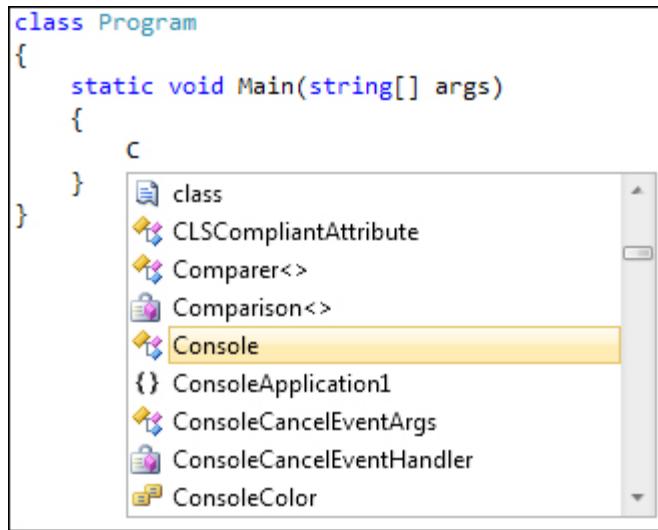
```
class Program
{
    static void Main(string[] args)
    {
        |
    }
}
```

The cursor automatically indents for you, ready to type something. Note where the curly brackets are, though, in the code above. You have a pair for **class Program**, and a pair for the Main method. Miss one out and you'll get error messages.

The single line of code we'll write is this (but don't write it yet):

```
Console.WriteLine("Hello C Sharp!");
```

First, type the letter "C". You'll see a popup menu. This popup menu is called the IntelliSense menu. It tries to guess what you want, and allows you to quickly add the item from the list. But it should look like this, after you have typed the letters "Con" (the list will be styled differently, depending on your version of the C# software):



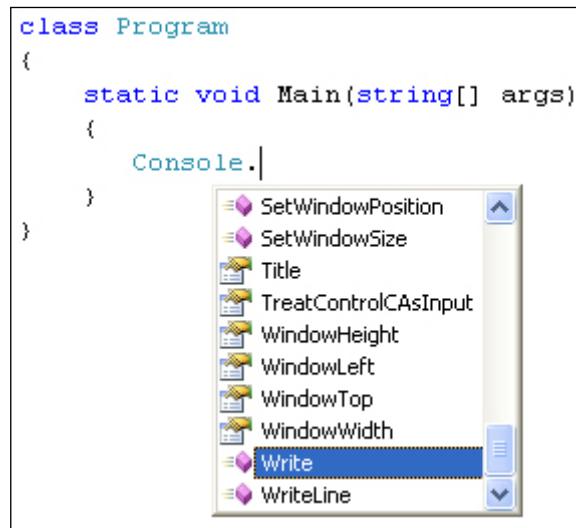
The icon to the left of the word Console on the list above means that it is a Class. But press the Enter key on your keyboard (you can also press the tab key). The word will be added to your code

A screenshot of a C# code editor window. The code in the editor is:

```
class Program
{
    static void Main(string[] args)
    {
        Console|
    }
}
```

The word 'Console' has been added to the code, and the cursor is positioned immediately after it.

Now type a full stop (period) immediately after the word Console. The Intellisense menu appears again:



You can use the arrow keys on your keyboard to move up or down the list. But if you type Write and then the letter "L" of Line, Intellisense will automatically move down and select it for you:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine
    }
}
```

The word `WriteLine` is highlighted in yellow, indicating it is a suggestion or part of a code completion dropdown.

Press the enter or tab key to add the word **WriteLine** to your code:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine
    }
}
```

Now type a left round bracket. As soon as you type the round bracket, you'll see something like this:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine()
    }
}
```

A tooltip box appears over the cursor, showing the method signature and a brief description: "1 of 19 void **Console.WriteLine()** Writes the current line terminator to the standard output stream."

`WriteLine` is another Method (a Method is just some code that does a particular job). But the box is telling you that there are 19 different versions of this Method. You could click the small arrows to move up and down the list. Instead, type the following:

“Hello C Sharp!”

Don't forget the double quotes at the start and end (and don't copy and paste the line above as you'll have the wrong type of double quotes for C#). These tell C# that you want text. Your code will look like this:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello C Sharp!")
    }
}
```

Now type a right round bracket:

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello C Sharp!");
    }
}

```

Notice the red wiggly line at the end. This is the coding environment's way of telling you that you've missed something out.

The thing we've missed out is a semicolon. All complete lines of code in C# must end with a semicolon. Miss one out and you'll get error messages. Type the semicolon at the end and the red wiggly line will go away. Your code should now look like this:

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello C Sharp!");
    }
}

```

Note all the different colours in your code. Visual C# colours the different parts of your code. The reddish colour between double quotes means that you want text; the green colour means it's a Class; blue words are ones that C# reserves for itself. These are called **Keywords**.

(If you want, you can change these colours. From the menu bar at the top, click **Tools > Options**. Under **Environment**, click **FONTs and Colors**.)

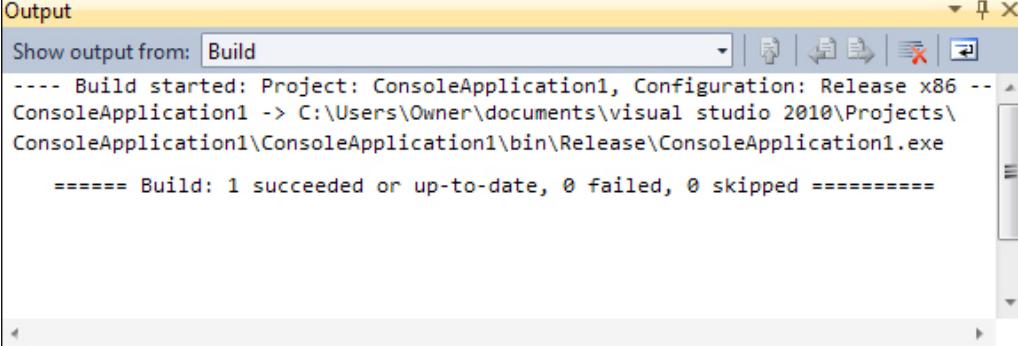
Time now to Build and Run your code!

Running your Programmes

You can test your programme a number of ways. First, it has to be built. This is when everything is checked to see if there are any errors. Try this:

- From the **View** menu at the top of Visual Studio Express, click **Output**. You'll see a window appear at the bottom (In C# 2010, if you can't see an Output entry, click the **Tools** menu. From the Tools menu, select **Settings > Expert Settings**. The Output menu item should then appear on the View menu.)

- From the Build menu at the top of Visual Studio Express, click **Build Solution**
- You should see the following report:



The screenshot shows the 'Output' window in Visual Studio. The title bar says 'Output'. The dropdown menu says 'Show output from: Build'. The main area displays the following text:
---- Build started: Project: ConsoleApplication1, Configuration: Release x86 ----
ConsoleApplication1 -> C:\Users\Owner\documents\visual studio 2010\Projects\ConsoleApplication1\ConsoleApplication1\bin\Release\ConsoleApplication1.exe
===== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ======

The final line is this:

Build: 1 succeeded or up-to-date, 0 failed, 0 skipped

That's telling you that everything is OK.

Now try this:

- Delete the semicolon from the end of your line of code
- Click **Build > Build Solution** again
- Examine the output window (version 2012 of C# will just show an error and won't build)

This time, you should see these two lines at the end of the report:

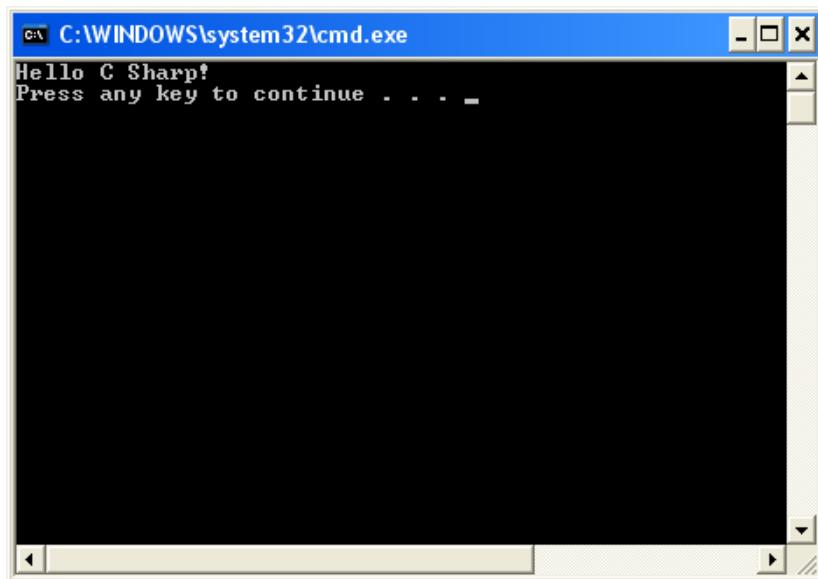
Compile complete -- 1 errors, 0 warnings
Build: 0 succeeded or up-to-date, 1 failed, 0 skipped

So it's telling you that it couldn't build your solution because there was 1 error.

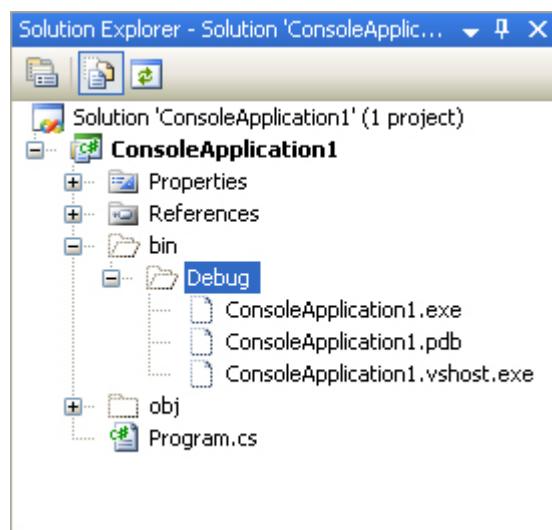
Put the semicolon back at the end of the line. Now click **Debug** from the menu at the top of Visual Studio Express. From the Debug menu, select **Start Debugging**.

You should see a black DOS window appear and then disappear. Your programme has run successfully!

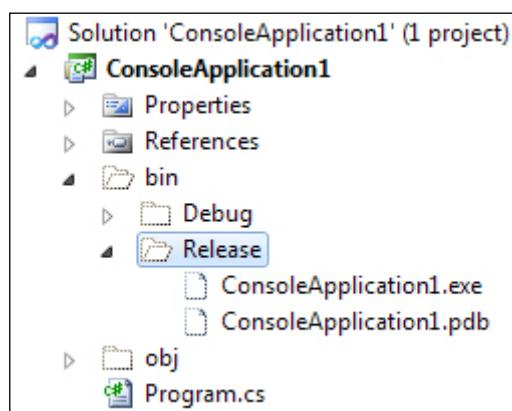
To actually see your line of text, click **Debug > Start Without Debugging**. You should now see this:



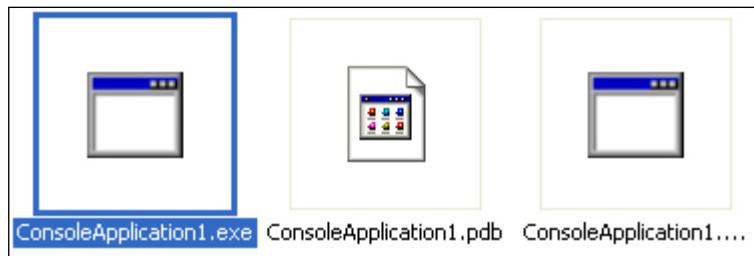
And that's your programme! Have a look at the Solution Explorer on the right. Because the project has been built, you'll see two more files under Debug:



However, in C# 2010 you'll see a **Release** folder. Expand this:



We now have a ConsoleApplication1.exe and ConsoleApplication1.pdb. The exe file is an executable programme, and it appears in the bin/debug folder. Switch back to Windows Explorer, if you still have it open. You'll see the **exe** file there (in the **Release** folder in C# 2010, but in the **Debug** folder in other versions):



You could, if you wanted, create a desktop shortcut to this exe file. When you double click the desktop shortcut, the programme will run.

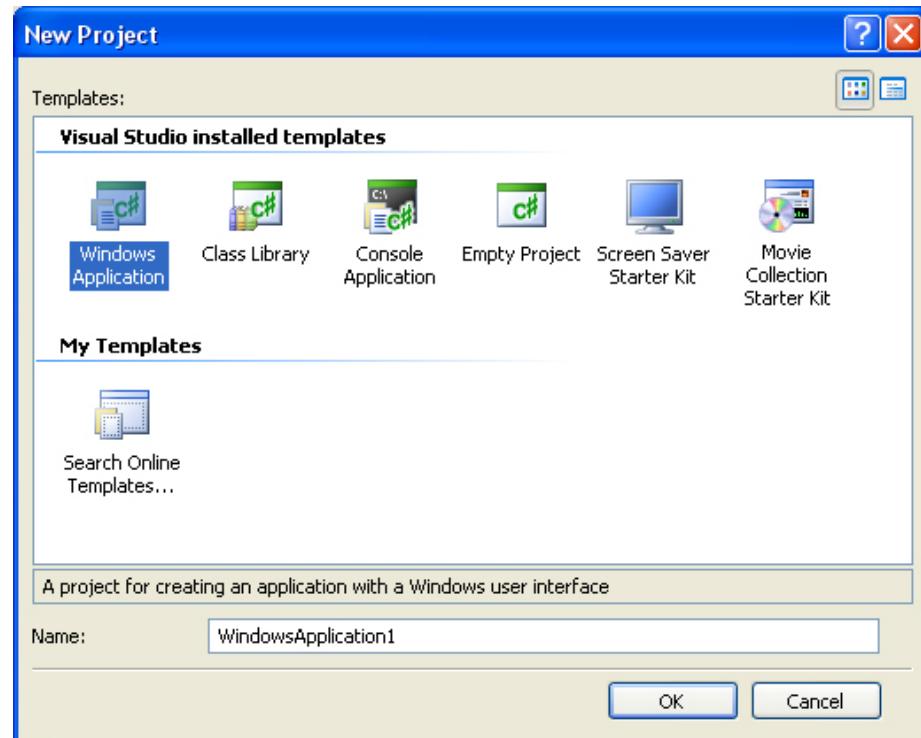
But that's enough of Console Applications – we'll move on to creating Windows Applications.

Your Windows First Form

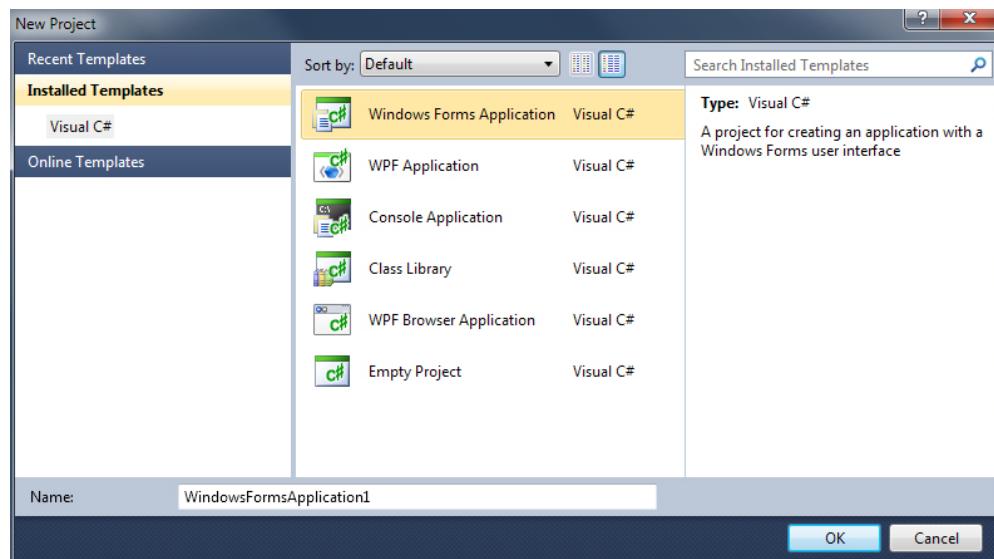
From now on, we're going to be creating Windows Forms Applications, rather than Console Applications. Windows Forms Applications make use of something called a Form. The Form is blank at first. You then add controls to your form, things like buttons, text boxes, menus, check boxes, radio buttons, etc. To get your first look at a Windows Form, do the following.

If you still have your Console Application open from the previous section, click **File** from the menu bar at the top of Visual Studio Express. From the File menu, click on **Close Solution**.

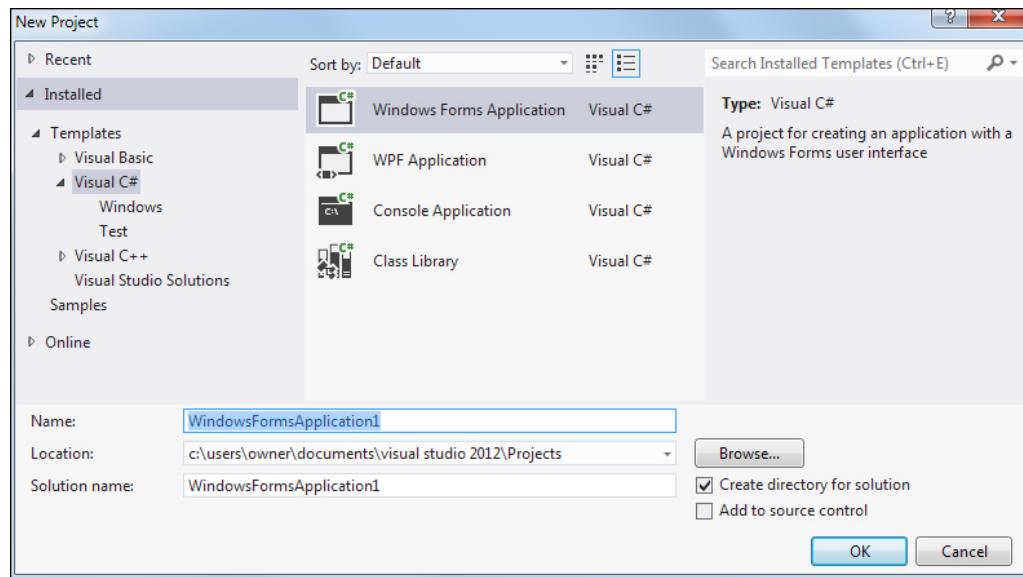
To create your first Windows form project, click the **File** menu again. This time, select **New Project** from the menu. When you do, you'll see the New Project dialogue box again:



Or this one in C# 2010:

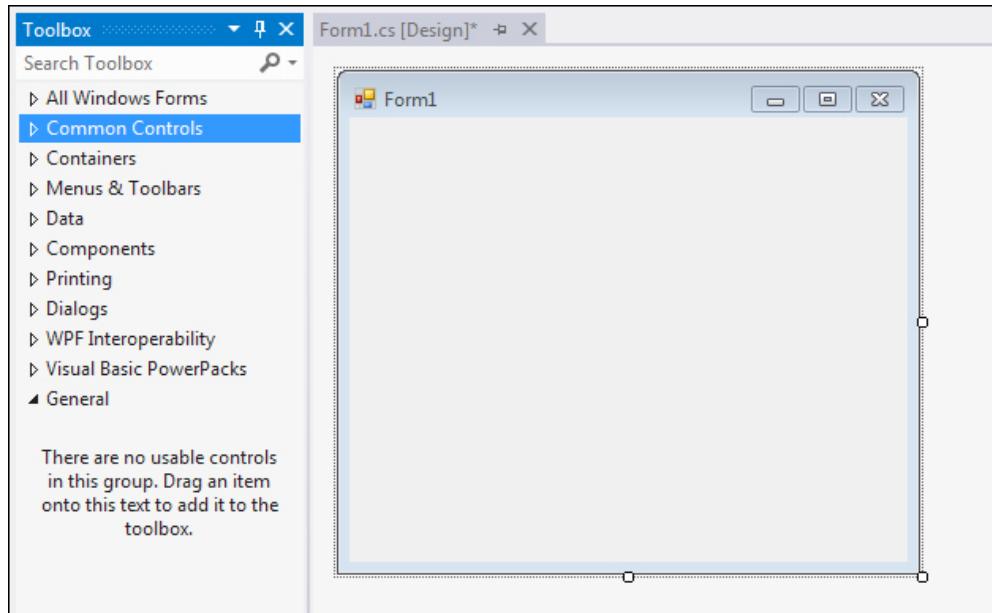


In Visual Studio Express 2012, click **Visual C#**, under **Templates** on the left:



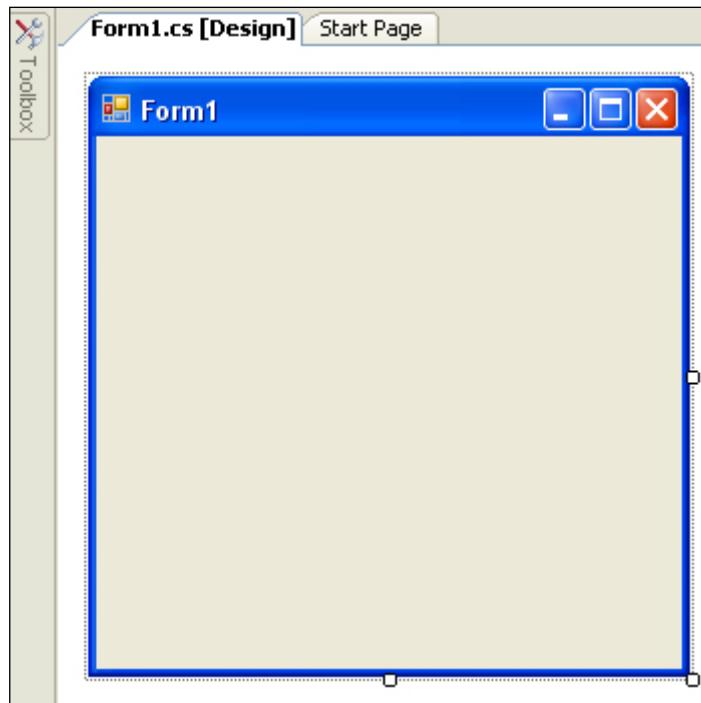
In all versions, select **Windows Forms Application** (or Windows Application in earlier versions) from the available templates. Keep the Name on the default of **WindowsFormsApplication1** and then click OK.

When you click OK, a new Windows Application project will be created for you:

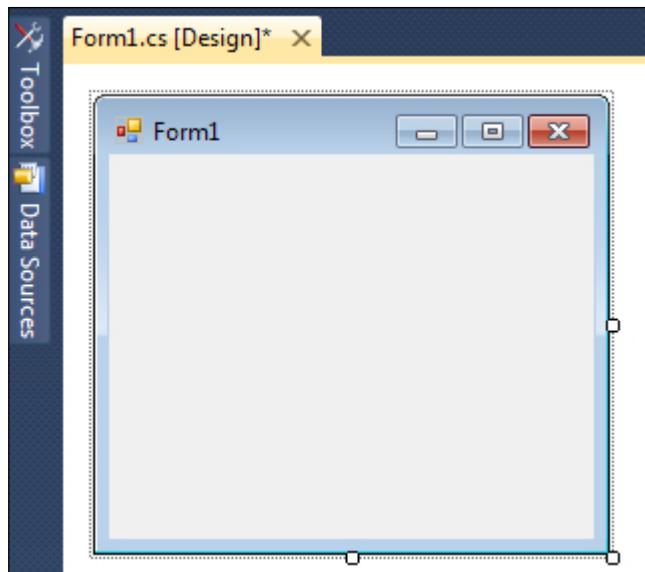


The obvious difference from the Console Application you created in the previous section is the blank Form in the main window. Notice the Toolbox, though, on the left hand side. We'll be adding controls from the Toolbox to that blank Form1 you can see in the image above.

If you can't see the Toolbox, you may just see the Tab, as in the following image:



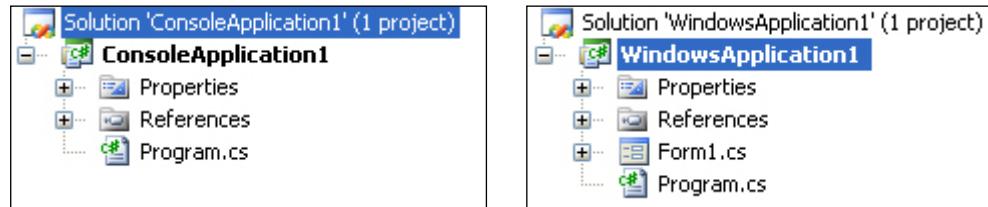
In C# 2010, your screen will look like this (version 2012 is similar but less colourful):



Move your mouse over to the Toolbox tab, or just click it. It will expand to show you the items in the toolbox. If you want to permanently display the Toolbox, click on the pin symbol:



Notice the Solution Explorer on the right side of your screen. (If you can't see the Solution Explorer, click its entry on the **View** menu at the top of Visual Studio Express.) If you compare it with the Solution Explorer when you created your Console Application, you'll see there's only one difference – the Form.



Console Application

Windows Application

We still have the Properties, the References and the Program.cs file. Double click the Program.cs file to open it, and you'll see some familiar code:

```

using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace WindowsApplication1
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}

```

And here's the code from the Console Application (newer versions of Visual Studio will have more **using** lines):

```

using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}

```

Both have **using** lines, a **namespace**, a **class** called **Program**, and a **Main Method**.

The **Main** Method is the entry point for your programme. The code between the curly brackets of **Main** will get executed when the programme first starts. The last line in the **WindowsApplication1** code above is the one that **Runs** Form1 when the **Application** starts.

You can do other things here. For example, suppose you had a programme that connects to a server. If it finds a connection then it loads some information from a database. In the Main Method, you could check that the server connection is OK. If it's not, display a second form; if it's OK, then display the first form.

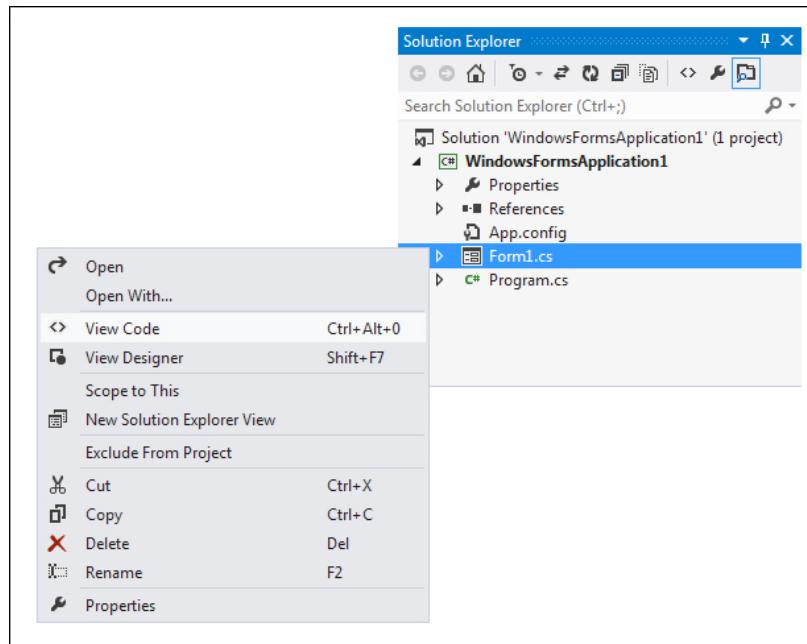
But don't worry if all that code has you scratching your head. The thing to bear in mind here is that a method called **Main** starts your programme. And **Program.cs** in the Solution Explorer on the right is where the code for **Main** lives.

But we won't be writing code in the **Program.cs** file, so we can close it. Have a look near the top of the coding window, and you'll see some tabs:

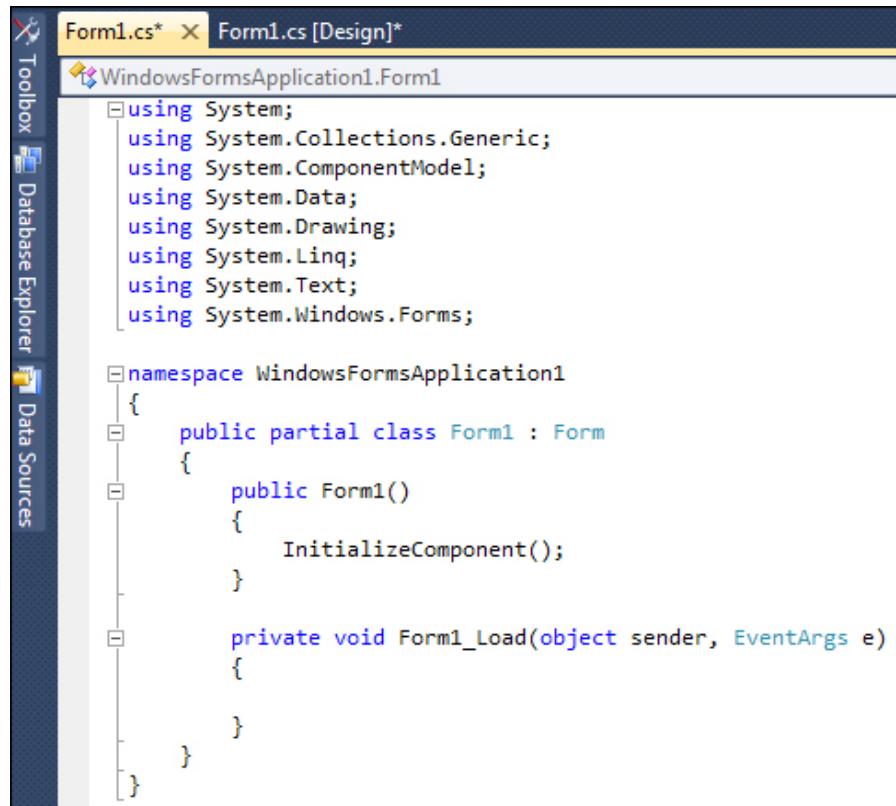


In version 2008, right click the **Program.cs** tab and select **Close** from the menu that appears. In newer versions, just click the X to close the tab. You should now see your form again (you may have a **Start** tab as well. You can close this, if you want).

To see the window where you'll write most of your code, right click **Form1.cs** in the Solution Explorer:

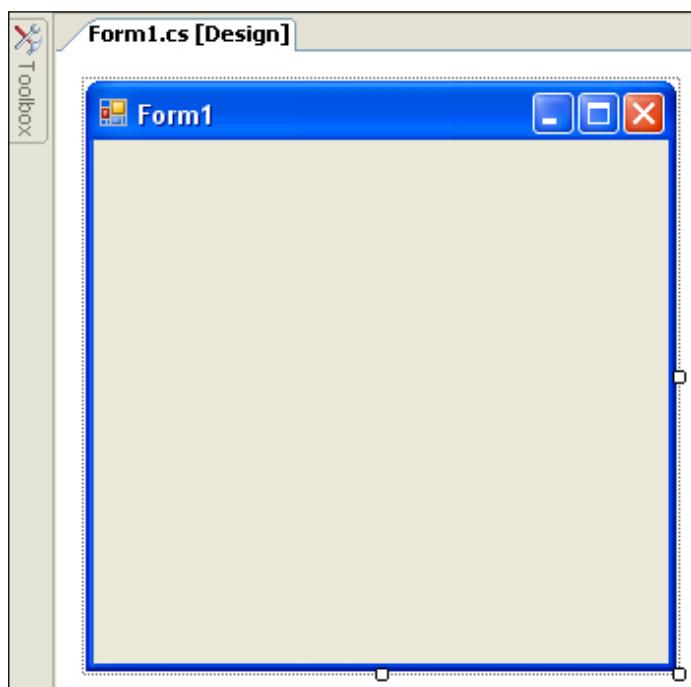


The menu has options for **View Code** and **View Designer**. The Designer is the Form you can see at the moment. Click **View Code** from the menu to see the following window appear (you can also press the F7 key on your keyboard in earlier versions, and CTRL + ALT + 0 in version 2012):



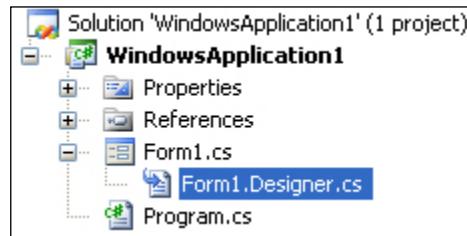
```
Form1.cs*  X  Form1.cs [Design]*  
WindowsFormsApplication1.Form1  
using System;  
using System.Collections.Generic;  
using System.ComponentModel;  
using System.Data;  
using System.Drawing;  
using System.Linq;  
using System.Text;  
using System.Windows.Forms;  
  
namespace WindowsFormsApplication1  
{  
    public partial class Form1 : Form  
    {  
        public Form1()  
        {  
            InitializeComponent();  
        }  
  
        private void Form1_Load(object sender, EventArgs e)  
        {  
        }  
    }  
}
```

This is the code for the Form itself. This Form (screenshot from version 2008):



The code has a lot more **using** statements than before. Don't worry about these for now. They just mean "using some code that's already been written".

The code also says **partial class Form1**. It's partial because some code is hidden from you. To see the rest of it (which we don't need to alter), click the plus or arrow symbol next to **Form1.cs** in the Solution Explorer:



Now double click **Form1.Designer.cs**. You'll see the following code:

```

namespace WindowsApplication1
{
    partial class Form1
    {
        /// <summary> ...
        private System.ComponentModel.IContainer components = null;

        /// <summary> ...
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        Windows Form Designer generated code
    }
}

```

Again, you see **partial class Form1**, which is the rest of the code. Click the plus symbol next to **Windows Form Designer generated code**. You'll see the following:

```

#region Windows Form Designer generated code

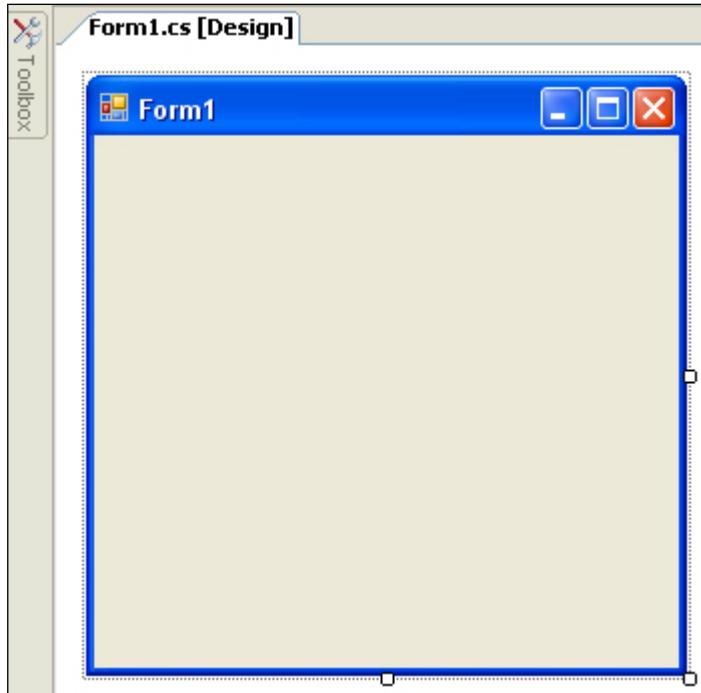
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
}

#endregion

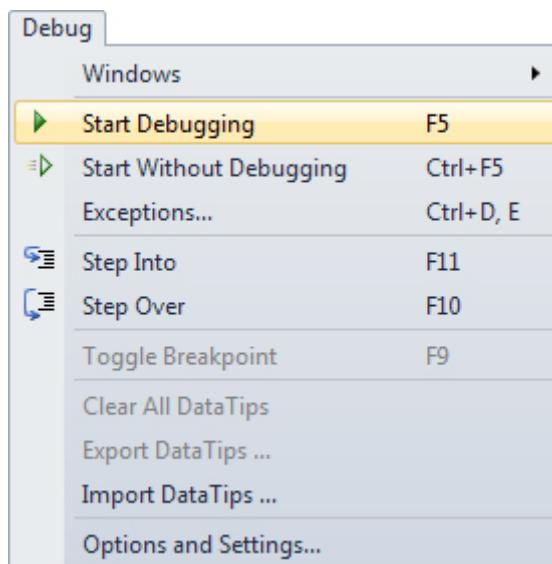
```

InitializeComponent is code (a Method) that is automatically generated for you when you create a new Windows Application project. As you add things like buttons and text boxes to your form, more code will be added here for you. But you don't need to do anything in this window, so you can right click the **Form1.Designer.cs** tab at the top, and click **Close** from the menu.

Click back on the **Form1.cs** tab at the top to see your form again. If the tab is not there, right click **Form1.cs** in the Solution Explorer on the right. From the menu, select **View Designer**. Here's what you should be looking at:



It's in Designer view that we'll be adding things like buttons and text boxes to our form. But you can run this programme as it is. From the **Debug** menu at the top, click **Start Debugging** (Or you can just press the F5 key on your keyboard.):



When you click Start Debugging, Visual C# will build the programme first, and then run it, if it can. If it can't run your programme you'll see error messages.

But you should see your form running on top of Visual Studio. It will have its own Red X, and its own minimize and maximize buttons. Click the Red X to close your programme, and to return to Visual Studio.

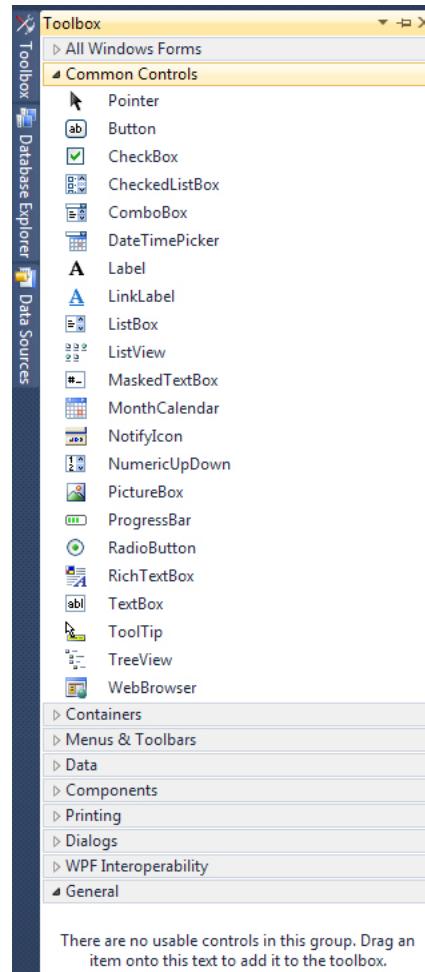
From now on, when we say Run your programme, this is what we mean: either press F5, or click **Debug > Start Debugging**.

OK, time for you to start adding things to the form, and to do a little coding!

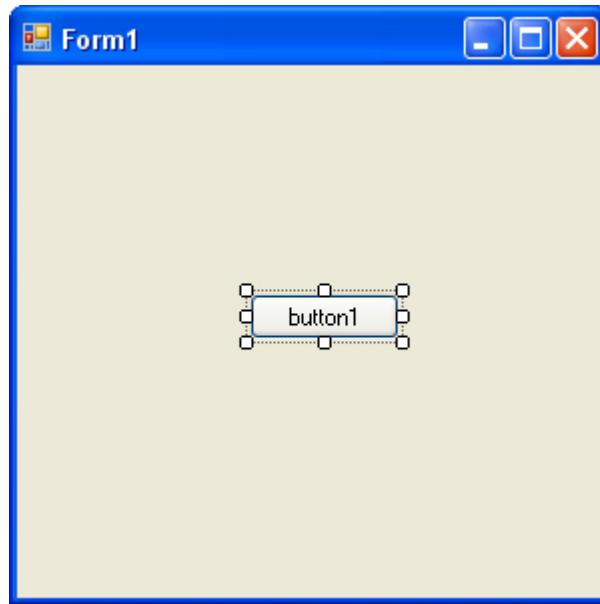
Adding Controls to a Blank Form

The first thing we'll do is to add a button to the blank form. We'll then write a single line of code, so that you can see how things work.

If you want to add a control to a form, you can use the Toolbox on the left of Visual Studio. Move your mouse over to the Toolbox, and click the arrow symbol (or plus symbol) next to **Common Controls**. You should see the following list of things that you can add to your form:



Click the **Button** item under the Common Controls heading. This will select it. Now click once anywhere on your form. A button will be drawn for you, and your Form will look like this:



(You can also hold down your left mouse button and drag out a button to the size you want it.)

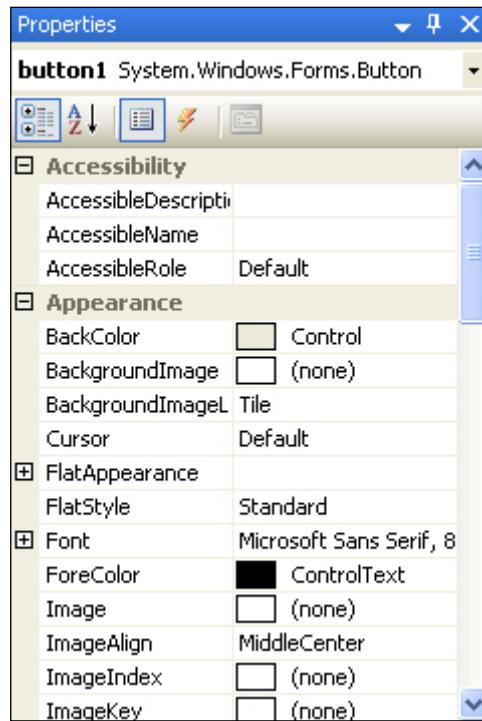
A button is something you want people to click on. When they do, the code you write gets executed. The text on the button, which defaults to “button1”, can be changed. You can add anything you like here, but it should be something that’s going to be useful for your users, such as “Open a text file”, or “Calculate Now”.

We’re going to display a simple message box when the button is clicked. So we need to add some text.

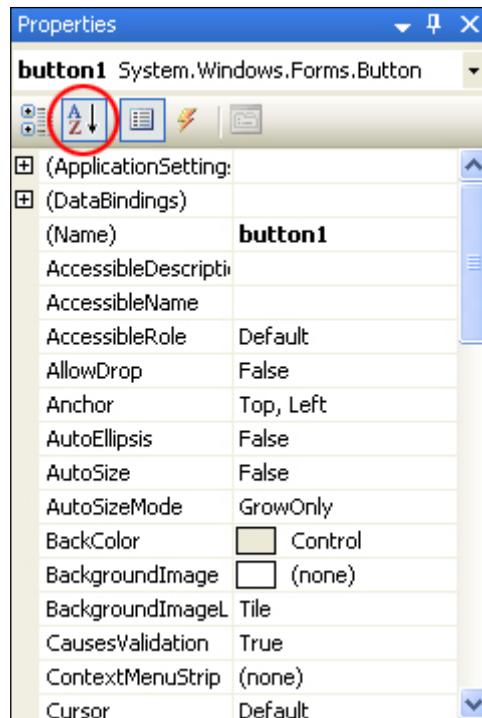
Properties of a Control

The controls you add to a form have something called **Properties**. A property of a control is things like its Height, its Width, its Name, its Text, and a whole lot more besides. To see what properties are available for a button, make sure the button is selected, as in the image above. If a control is selected, it will have white squares surrounding it. If your button is not selected, simply click it once.

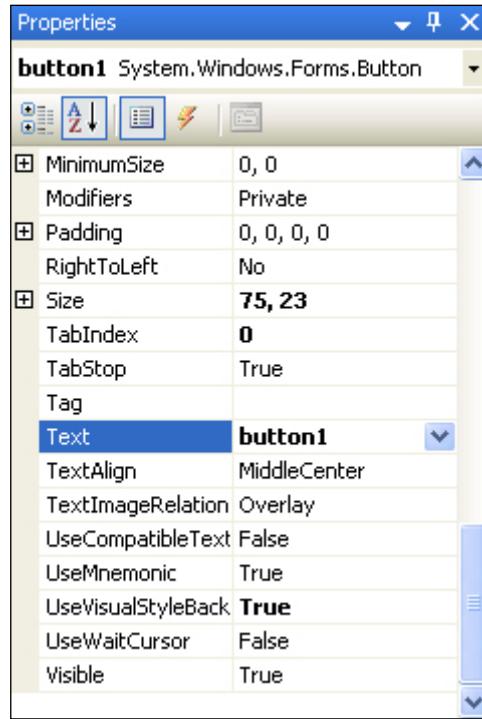
Now look in the bottom right of Visual C# Express, just below the Solution Explorer. You should see the Properties Window (if it’s not there, select it from the **View** menu at the top. Again, 2010 users need to click **Tools > Settings > Expert Settings** to see the full list of menu items.):



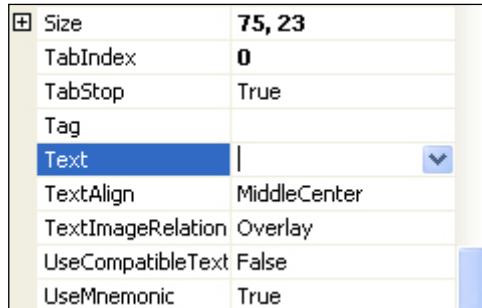
To view the list of Properties in alphabetical order, click the AZ symbol at the top, circled in red in the image below:



As you can see, there's a lot of Properties for a button. Scroll down to the bottom and locate the **Text** Property:



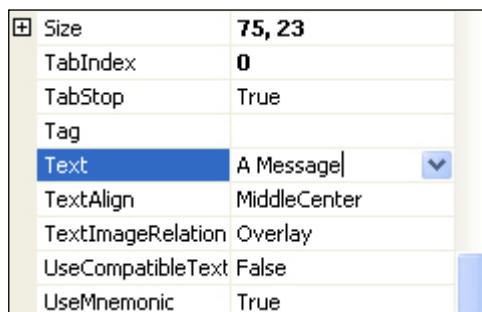
The Text Property, as its name suggests, is the Text you want to appear on the button. At the moment, it says button1. Click inside of the text area of button1. Delete the default text:



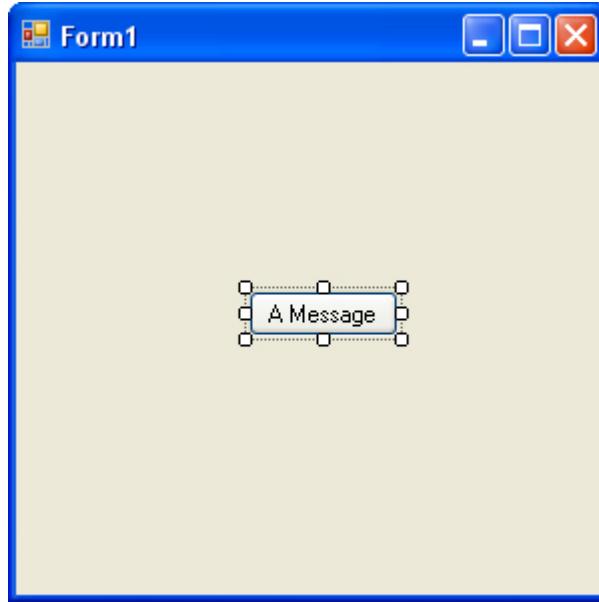
Now type the following:

A Message

The Text part of your Properties Window will then look like this:



Now press the enter key on your keyboard. Have a look at your Form, and the Text on the button should have changed:



There are a few more Properties we can change before we get to the code.

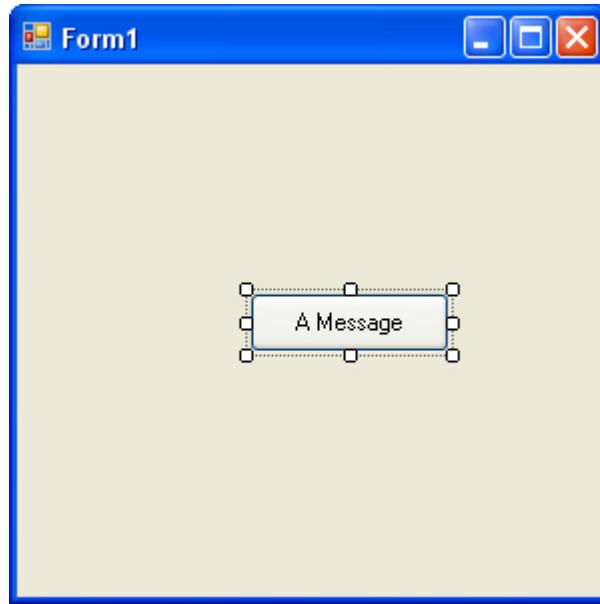
Locate **Size** in the Properties Window:

| | | |
|-----|-------------|---------------|
| [+] | Padding | 0, 0, 0, 0 |
| [+] | RightToLeft | No |
| [+] | Size | 75, 23 |
| [+] | TabIndex | 0 |
| [+] | TabStop | True |

The first number, 75, is the width of the button. The second number, 23, is the height of the button. The two numbers are separated by a comma. Change the numbers to 100, 30:

| | | |
|-----|-------------|------------------|
| [+] | Modifiers | Private |
| [+] | Padding | 0, 0, 0, 0 |
| [+] | RightToLeft | No |
| [+] | Size | 100, 30 |
| [+] | TabIndex | 0 |
| [+] | TabStop | True |
| [+] | Tag | |
| [+] | Text | A Message |

Press the enter key again, and the size of your button will change:



You can move your button around the Form by clicking it with the left mouse button to select it. Hold down your left mouse button and drag your button around the form. Let go of your left mouse button when you're happy with the new location.

Exercise

You can also move a button by changing its **Location** property. Use the Properties Window to change the Location of your button to a position on the form of 100, 50. (100 means 100 units from the left edge of the form; 50 means 50 units down from the top of the form.)

Exercise

A Form also has lots of Properties. Click away from the button and on to the Form itself. The Properties for the form will appear in the Properties Window. Change the **Text** Property of the Form to **A First Message**.

Exercise

The Form, like the button, also has a Size Property. Change the Size of the Form to 300, 200.

After you complete the three exercises above, your Form should look like this:



When you changed the Text property of the Form, you changed the text that runs across the blue bar at the top, the text in white. You can type anything you like here, but it should be something that describes what the form is all about or what it does. Often, you'll see the name of the software here, like Microsoft Word, or Adobe Acrobat.

We can now move on to some code, though, and then run the Form to see what it all looks like.

Adding code to a Button

What we want to do is to display a message box whenever the button is clicked. So we need the coding window. To see the code for the button, double click the button you added to the Form. When you do, the coding window will open, and your cursor will be flashing inside of the button code. It will look like this:

```
Form1.cs Form1.cs [Design] Start Page
WindowsApplication1.Form1 button1_Click(object sender, EventArgs e)

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
        }
    }
}
```

The only thing different from the last time you saw this screen is the addition of the code for the button. This code:

```
private void button1_Click(object sender, EventArgs e)
{
}
```

This is just another Method, a piece of code that does something. The name of the Method is **button1_Click**. It's called button1 because that's currently the Name of the button. When you changed the Text, Location, and Size properties of the button, you could have also changed the **Name** property from button1 (the default Name) to something else.

The `_Click` part after `button1` is called an **Event**. Other events are `MouseDown`, `LocationChanged`, `TextChanged`, and lots more. You'll learn more about Events later.

After `_Click`, and in between a pair of round brackets, we have this:

`object sender, EventArgs e`

These two are known as arguments. One argument is called **sender**, and the other is called **e**. Again, you'll learn more about arguments later, so don't worry about them for now.

Notice that there is a pair of curly brackets for the button code:

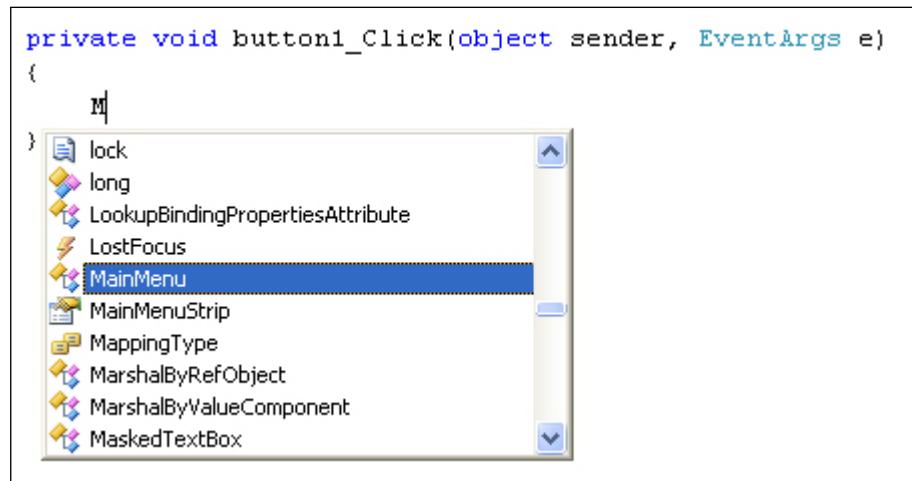
```
private void button1_Click(object sender, EventArgs e)
{
}
```

If you want to write code for a button, it needs to go between the two curly brackets. We'll add a single line of code.

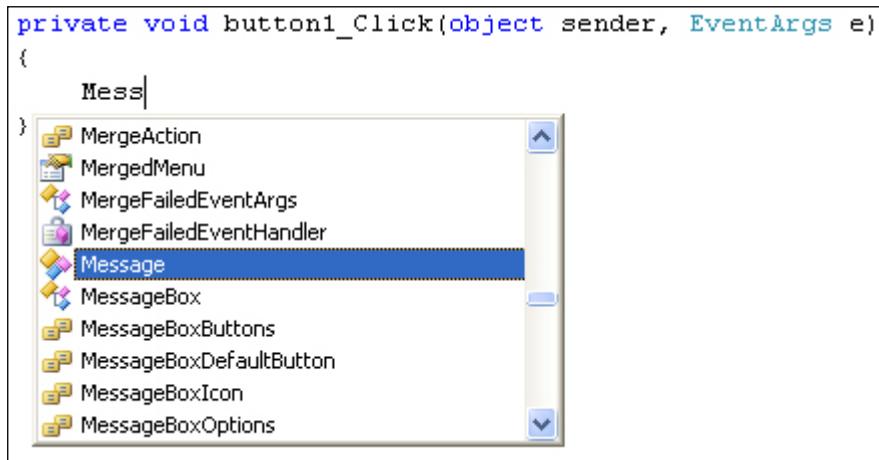
A MessageBox

We want to display a message box, with some text on it. This is quite easy to do in C#.

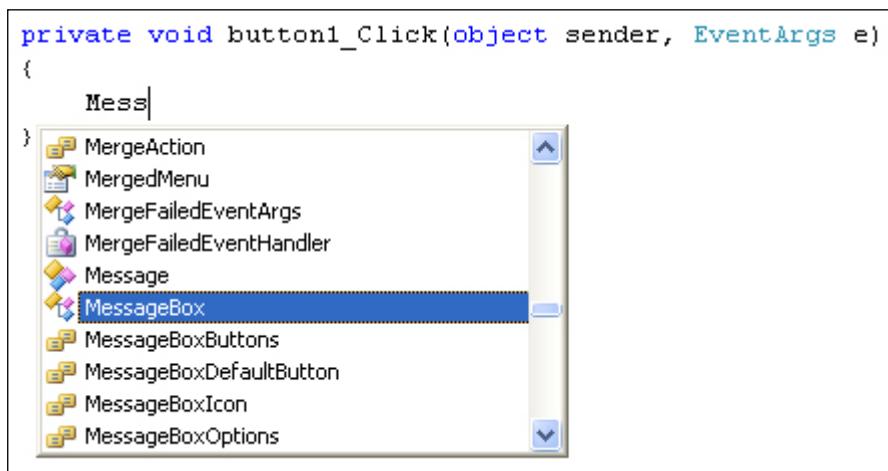
Position your cursor between the two curly brackets. Then type a capital letter "M". You'll see the IntelliSense list appear:



Now type “ess” after the “M”. IntelliSense will jump down:



The only options that start with Mess are all Message ones. The one we want is MessageBox. You can either just type the rest, or even easier is to press the down arrow on your keyboard to move down to MessageBox:



When you have MessageBox selected, hit the enter key on your keyboard (or double click the entry on the list, or press the tab key). The code will be added for you:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox|
}
```

Now type a full stop (period) after the “x” of **MessageBox**. The IntelliSense list will appear again:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.|
```

The IntelliSense list shows three methods: Equals, ReferenceEquals, and Show. The Show method is highlighted with a blue selection bar.

There are only three items on the list now, and all Methods (you can tell they are Methods because they have the purple block icon next to them) Double click on **Show**, and it will be added to your code:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show|
}
```

Because **Show** is a Method, we need some round brackets. The text for our message box will go between the round brackets. So type a left round bracket, just after the “w” of “Show”:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show(|
```

The IntelliSense list shows the **Show** method with a parameter **text** highlighted. A tooltip below the list says: "1 of 21 ▾ DialogResult MessageBox.Show (string text)
text: The text to display in the message box."

As soon as you type the left round bracket after the “w”, you’ll see all the different ways that the Show method can be used. There are 21 different ways in total. Fortunately, you don’t have to hunt through them all! Type the following, after the left round bracket: (Don’t forget the double quotation marks. But don’t copy and paste below, because the quote marks are a different style from the ones C# uses.)

“My First Message”

After the final double quote mark, type a right round bracket. Then finish off the line by typing a semi-colon (;), Your coding window will then look like this:

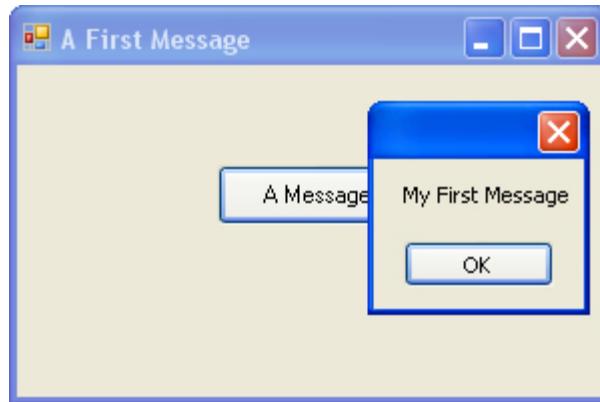
```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("My First Message");
}
```

The text in the dark reddish colour is what will be displayed in your message box. To try it out, save your work by clicking **File** from the menu bar at the top of Visual Studio. From the File menu, click **Save All**. You'll then see the same **Save** box you saw for the Console Application. Save the project.

Run your programme by clicking **Debug > Start Debugging**. Or just press the F5 key on your keyboard. Your programme will look like this:



Click your button to see your Message Box:



Congratulations! It's your first message!

Other things to do with the Message Box

If you look at the message box in the image above, you'll notice there's no Title in the blue area to the left of the red X – it's blank. You can add a Title quite easily.

Click OK on your Message Box. Then click the Red X on your programme to exit it. This will return you to Visual Studio. Go back to the coding window (press F7 on your keyboard, if you can't see it).

Position your cursor after the final double quote of "My First Message", circled in red in the image below:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("My First Message");
}
```

Now type a comma. As soon as you type a comma, you'll see the list of **Show** options again:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("My First Message",);
}
▲ 2 of 21 ▾ DialogResult MessageBox.Show (IWin32Window owner, string text)
Displays a message box in front of the specified object and with the specified text.
```

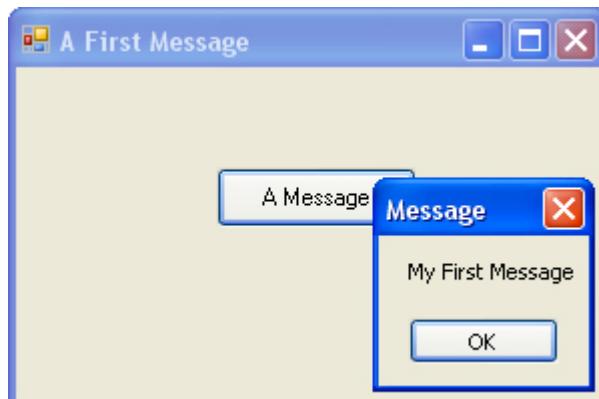
Type the following:

"Message"

Again, you need the double quotes. But your line of code should look like this:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("My First Message", "Message");
}
```

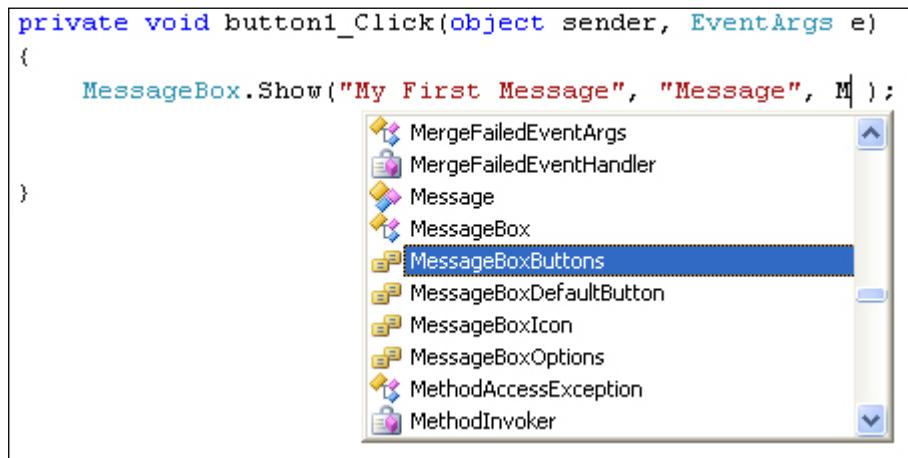
When your line of code looks like the one above, Run your programme again. Click your button and you should see a Title on your Message Box:



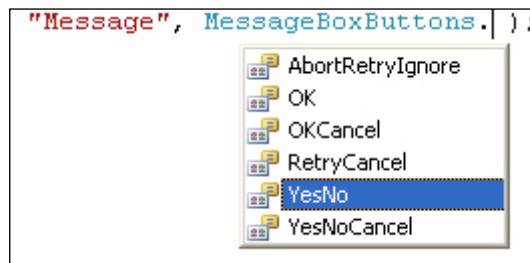
Other Button Options

Rather than having just an OK button, you can add buttons like Yes, No, and Cancel. We'll add a **Yes** and a **No** button.

Return to your coding window. After the second double quote of the Title you've just added, type another comma. Hit the spacebar on your keyboard once, and you'll see the IntelliSense list appear. (If it doesn't appear, just type a capital letter "M").



The one that adds buttons to a message box is, you won't be surprised to hear, **MessageBoxButtons**. Press the enter key on your keyboard when this option is highlighted. It will be added to your code. Now type a full stop (period) after the final "s" of `MessageBoxButtons`. You'll see the button options:



Double click the one for **YesNo**, and it will be added to your code.

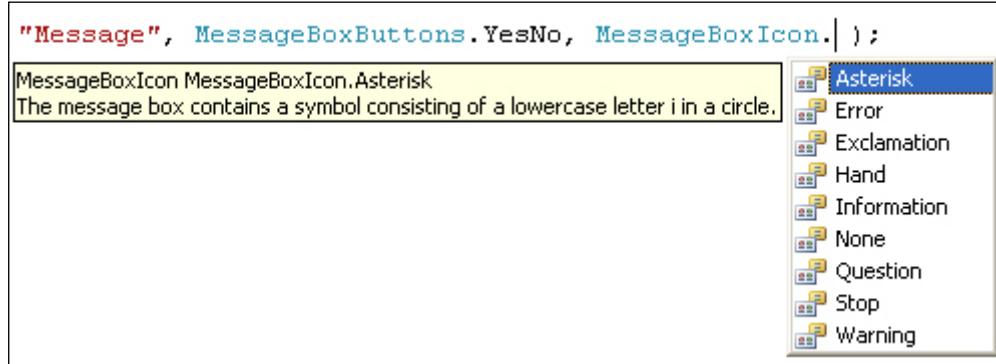
Run your programme again, and click your button. Your Message Box will then look like this:



Adding Icons to a Message Box

Another thing you can add to brighten up your Message Box is an Icon. It's easier to see what these are than to explain!

Type another comma after **MessageBoxButtons.YesNo**. After the comma, type a capital letter "M" again. From the IntelliSense list that appears, double click **MessageBoxIcon**. After MessageBoxIcon, type a full stop to see the available icons:



We've gone for **Asterisk**. Double click this to add it to your code. Run your programme again to see what the icon looks like on your Message Box:



Looks pretty impressive, hey! And all that with one line of code!

We'll move on to the important subject of variables, in the next part. First, try this Exercise.

Exercise

Try the other icons on the IntelliSense list, and see what they look like when your programme runs. Does the **Information** icon differ from the **Asterisk**? (To quickly display the IntelliSense list again, delete the word **Asterisk** from your code, then delete the full stop. Type the full stop again, and the IntelliSense list will reappear.)

Variables - Strings

Programmes work by manipulating data stored in memory. These storage areas come under the general heading of Variables. In this section, you'll see how to set up and use variables. You'll see how to set up both text and number variables. By the end of this section, you'll have written a simple calculator programme. We'll start with something called a string variable.

String Variables

The first type of variable we'll take a look at is called a string. String variables are always text. We'll write a little programme that takes text from a text box, store the text in a variable, and then display the text in a message box.

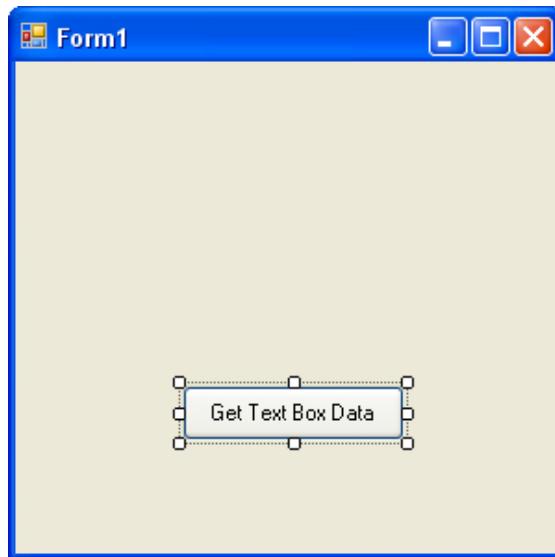
But bear in mind that a variable is just a storage area for holding things that you'll need later. Think of them like boxes in a room. The boxes are empty until you put something in them. You can also place a sticker on the box, so that you'll know what's in it. Let's look at a programming example.

If you've got your project open from the previous section, click **File** from the menu bar at the top of Visual Studio. From the File menu, click **Close Solution**. Start a new project by clicking File again, then **New Project**. From the New Project dialogue box, click on **Windows Forms Application**. In the Name box at the bottom, type **String Variables**.

Click OK, and you'll see a new form appear. Add a button to the form, just like you did in the previous section. Click on the button to select it (it will have the white squares around it), and then look for the Properties Window in the bottom right of Visual Studio. Set the following Properties for your new button:

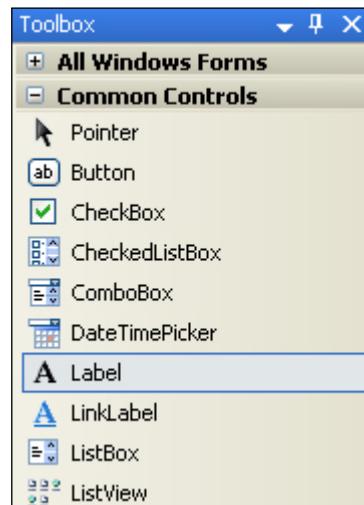
| | |
|------------------|-------------------|
| Name: | btnStrings |
| Location: | 90, 175 |
| Size: | 120, 30 |
| Text: | Get Text Box Data |

Your form should then look something like this:



We can add two more controls to the form, a Label and a Text Box. When the button is clicked, we'll get the text from the text box and display whatever was entered in a message box.

A Label is just that: a means of letting your users know what something is, or what it is for. To add a Label to the form, move your mouse over to the Toolbox on the left. Click the **Label** item under **Common Controls**:



Now click once on your form. A new label will be added:

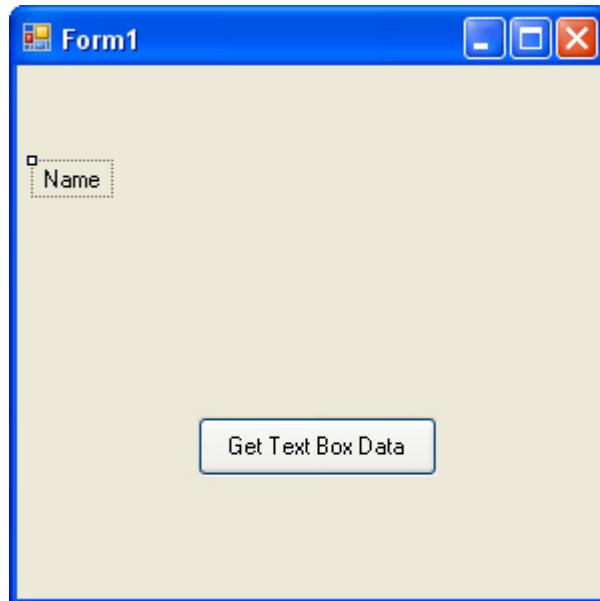


The Label has the default text of **label1**. When your label is selected, it will have just the one white square in the top left. When it is selected, the Properties Window will have changed. Notice that the properties for a label are very similar to the properties for a button – most of them are the same!

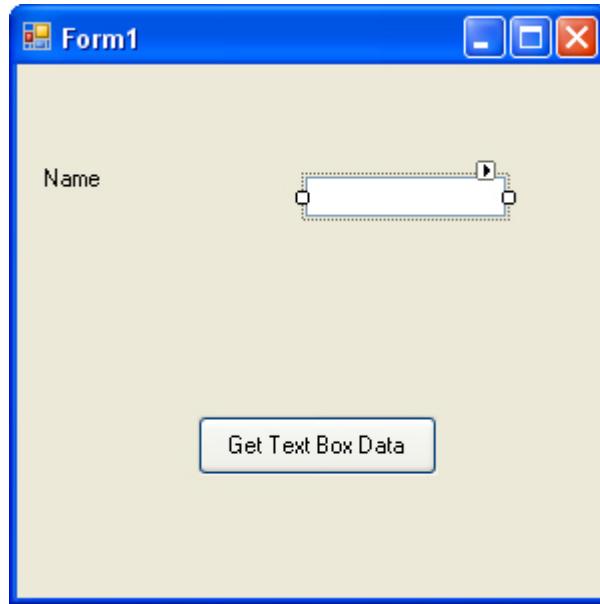
Change the following properties of your label, just like you did for the button:

Location: 10, 50
Text: Name

You don't really need to set a size, because Visual Studio will automatically resize your label to fit your text. But your Form should look like this:

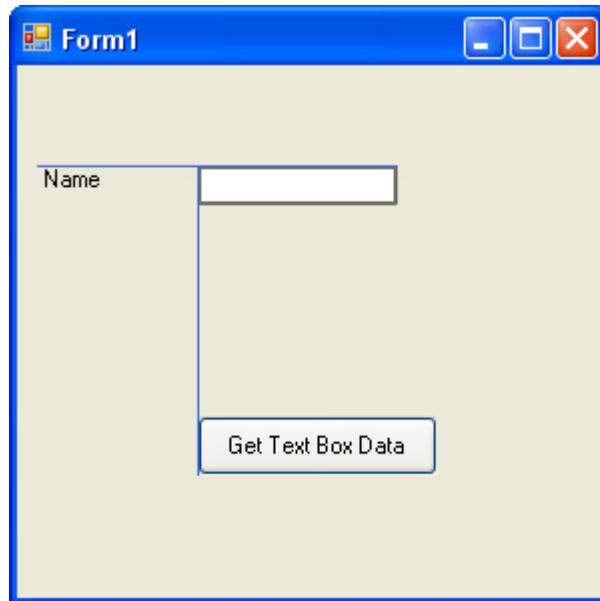


Move your mouse back over to the Toolbox. Click on the **TextBox** entry. Then click on your form. A new Text Box will be added, as in the following image:



Instead of setting a location for your text box, simply click it with your left mouse button. Hold your left mouse button down, and then drag it just to the right of the Label.

Notice that when you drag your text box around, lines appear on the form. These are so that you can align your text box with other controls on the form. In the image below, we've aligned the text box with the left edge of the button and the top of the Label.



OK, time to add some code. Before you do, click **File > Save All** from the menu bar at the top of Visual Studio. You can also run your programme to see what it looks like. Type some text in your text box, just to see if it works. Nothing will happen when you click your button, because we haven't written any code yet.

Let's do that now. Click the red X on your form to halt the programme, and you'll be returned to Visual Studio.

Code for the Button

Double click your button to open up the coding window. Your cursor will be flashing inside of the curly brackets for the button code:

```
■ using System;
| using System.Collections.Generic;
| using System.ComponentModel;
| using System.Data;
| using System.Drawing;
| using System.Text;
└ using System.Windows.Forms;

■ namespace String_Variables
{
    ■ public partial class Form1 : Form
    {
        ■ public Form1()
        {
            InitializeComponent();
        }

        ■ private void btnStrings_Click(object sender, EventArgs e)
        {
            |
        }
    }
}
```

Notice all the minus symbols on the left hand side. You can click these, and it will hide code for you. Click the minus symbol next to `public Form1()`. It will turn into a plus symbol, and the code for just this Method will be hidden:

```
■ using System;
| using System.Collections.Generic;
| using System.ComponentModel;
| using System.Data;
| using System.Drawing;
| using System.Text;
└ using System.Windows.Forms;

■ namespace String_Variables
{
    ■ public partial class Form1 : Form
    {
        ■ public Form1()...
        ■ private void btnStrings_Click(object sender, EventArgs e)
        {
            |
        }
    }
}
```

Hiding code like this makes the rest of the coding window easier to read.

Back to the button code, though. We're going to set up a string variable. To do this, you need two things: the Type of variable you want, and a name for your variable.

Click inside the two curly brackets of the button code, and add the following:

```
string firstName;
```

After the semi-colon, press the enter key on your keyboard to start a new line. Your coding window will then look like this:

```
namespace String_Variables
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            private void btnStrings_Click(object sender, EventArgs e)
            {
                string firstName;
            }
        }
    }
}
```

What you have done is to set up a variable called **firstName**. The Type of variable is a **string**. (Other types are **int**, **float**, and **Double**. These are all number variables that you'll meet shortly.) When you set up a variable, you always need to tell C# which variable Type you want. Since we're going to be getting text from a text box, we need the **string** Type.

After you have told C# which Type of variable you want, you then need to come up with a name for your variable. This is like the sticker on an empty box. The empty box is the variable Type. Think of these empty boxes as being of different sizes and different materials. A big, cardboard box is totally different from a small wooden one! But what you are really doing here is telling C# to set aside some memory, and that this storage area will hold strings of text. You give it a unique name so as to tell it apart from other items in memory. After all, would you be able to find the correct box, if they were all the same size, the same shape, the same colour, and had no stickers on them?

The name you pick for your variables, `firstName` in our case, can be almost anything you want – it's entirely up to you what you call them. But you should pick something that is descriptive, and gives you a clue as to what might be in your variable.

We say you can call your variable almost anything. But there are some rules, and some words that C# bags for itself. The words that C# reserves for itself are called

Keywords. There are 77 of these words, things like “using”, “for”, “new”, and “public”. If the name you have chosen for your variable turns blue in the coding window, then it’s a reserved word, and you should pick something else.

The only characters that you can use in your variable names are letters, numbers, and the underscore character (_). And you must start the variable name with a letter, or underscore. You’ll get an error message if you start your variable names with a number. So these are OK:

```
firstName
first_Name
firstName2
```

But these are not:

| | |
|------------------------|--------------------------------------|
| 1firstName | (Starts with a number) |
| first_Name& | (Ends with an illegal character) |
| first Name | (Two words, with a space in between) |

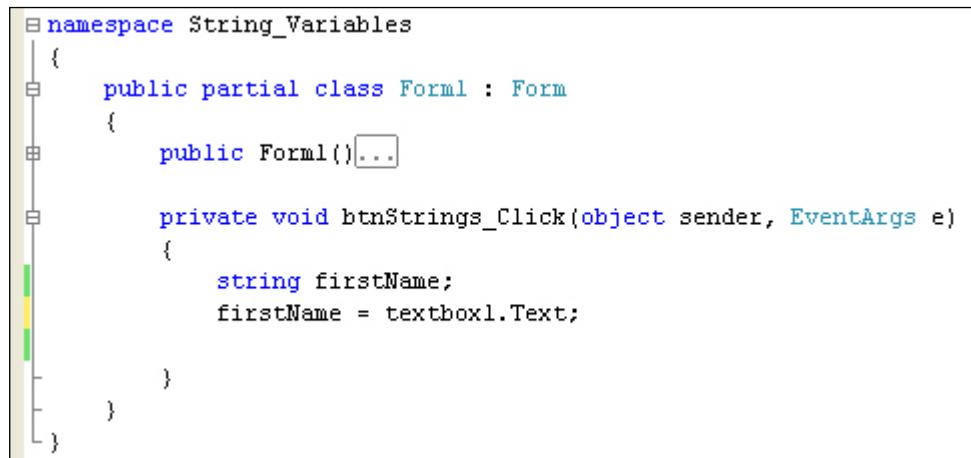
Notice that all the variable names above start with a lowercase letter. Because we’re using two words joined together, the second word starts with an uppercase letter. It’s recommended that you use this format for your variables (called camelCase notation.) So **firstName**, and not **Firstname**.

After setting up your variable (telling C# to set aside some memory for you), and giving it a name, the next thing to do is to store something in the variable. Add the following line to your code (don’t forget the semi-colon on the end):

```
firstName = textbox1.Text;
```

In later versions of C#, you may see a red or blue underline for `textbox1`. If so, just ignore it for now. You'll see why.

Your coding window will then look like this:

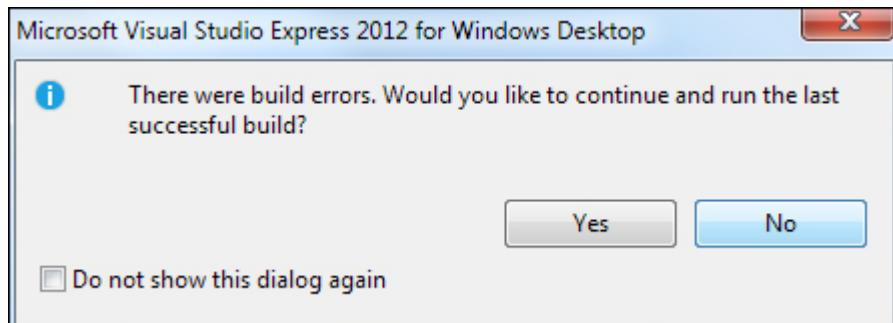


```
namespace String_Variables
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btnStrings_Click(object sender, EventArgs e)
        {
            string firstName;
            firstName = textbox1.Text;
        }
    }
}
```

To store something in a variable, the name of your variable goes on the left hand side of an equals sign. After an equals sign, you type what it is you want to store in the variable. For us, this is the **Text** from **textbox1**.

Except, there's a slight problem. Try to run your code. You should see an error message like this one:



Click **No**, and have a look at your code:

```
private void btnStrings_Click(object sender, EventArgs e)
{
    string firstName;
    firstName = textbox1.Text;
}
```

There is a blue wiggly line under **textbox1**. Hold your mouse over this and Visual Studio will tell you that:

The name ‘textbox1’ does not exist in the current context.

If you see an error like this, which is quite common, it means that Visual Studio cannot find anything with the name you've just typed. So it thinks we don't have a textbox called **textbox1**. And we don't! It's called **textBox1**. We've typed a lowercase "b" when it should be an uppercase "B". So it's important to remember that C# is case sensitive. This variable name:

firstName

is different to this variable name:

FirstName

The first one starts with a lowercase "f" and the second one starts with an uppercase "F".

Delete the lowercase "b" from your code and type an uppercase "B" instead. Run your programme again and you won't see the error message. Now stop your programme and return to the coding window. The blue wiggly line will have disappeared.

What have so far, then, is the following:

```
string firstName;  
firstName = textBox1.Text;
```

The first line sets up the variable, and tells C# to set aside some memory that will hold a string of text. The name of this storage area will be **firstName**.

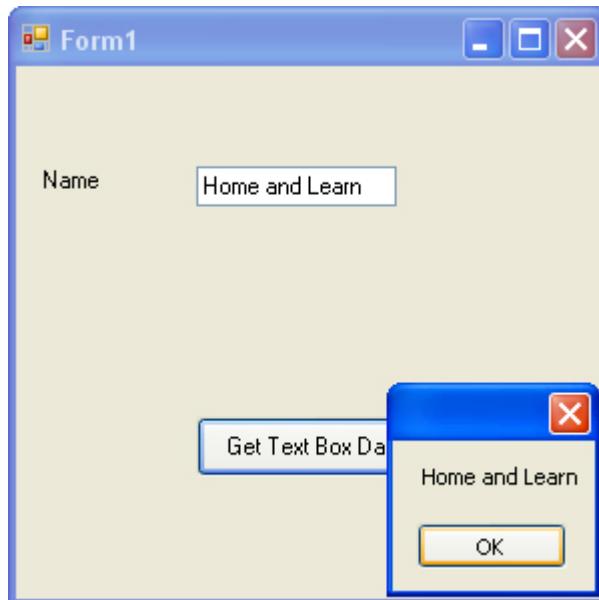
The second line is the one that actually stores something in the variable – the Text from a text box called **textBox1**.

Now that we have stored the text from the text box, we can do something with it. In our case, this will be to display it in a message box. Add this line to your code:

```
MessageBox.Show( firstName );
```

The `MessageBox.Show()` Method is one you've just used. In between the round brackets, you can either type text surrounded by double quotes, or you can type the name of a string variable. If you're typing the name of a variable, you leave the double quotes off. You can do this because C# knows what is in your variable (you have just told it on the second line of your code.)

Run your programme again. Type something in your text box, and then click the button. You should see the text you typed:



Halt your programme and return to the coding window.

Assigning text to a String Variable

As well as assigning text from a text box to your variable, you can assign text like this:

```
firstName = "Home and Learn";
```

On the right hand side of the equals sign, we now have some direct text surrounded by double quotes. This then gets stored into the variable on the left hand side of the equals sign. To try this out, add the following two lines just below your MessageBox line:

```
firstName = "Home and Learn";
MessageBox.Show(firstName);
```

Your coding window will then look like this:

```
private void btnStrings_Click(object sender, EventArgs e)
{
    string firstName;
    firstName = textBox1.Text;
    MessageBox.Show(firstName);

    firstName = "Home and Learn";
    MessageBox.Show(firstName);
}
```

Run your programme again. Type something in the text box, your own first name. Then click the button. You should see two message boxes, one after the other. The first one will display your first name. But the second will display “Home and Learn”.

We’re using the same variable name, here: **firstName**. The first time we used it, we got the text directly from the text box. We then displayed it in the Message Box. With the two new lines, we’re typing some text directly in the code, Home and Learn, and then assigning that text to the **firstName** variable. We’ve then added a second **MessageBox.Show()** method to display whatever is in the variable.

Concatenation

Another thing we can do is something called Concatenation. Concatenation is joining things together. You can join direct text with variables, or join two or more variables to make a longer string. A coding example may clear things up.

Delete the two new lines you’ve just added. Now add a second variable, just below the first one:

```
string messageText;
```

So your coding window should look like this:

```
private void btnStrings_Click(object sender, EventArgs e)
{
    string firstName;
    string messageText;

    firstName = textBox1.Text;
    MessageBox.Show(firstName);
}
```

We want to store some text inside of this new variable, so add the following line of code just below **string messageText**:

```
messageText = "Your name is: ";
```

Your code window will then look like ours below:

```
private void btnStrings_Click(object sender, EventArgs e)
{
    string firstName;
    string messageText;

    messageText = "Your name is: ";

    firstName = textBox1.Text;
    MessageBox.Show(firstName);
}
```

When the message box displays, we want it to say something like “You name is John”. The variable we’ve called **messageText** holds the first part of the string, “Your name is ”. And we’re getting the person’s name from the text box:

```
firstName = textBox1.Text;
```

The person’s name is being stored in the variable called **firstName**. To join the two together (concatenate) C# uses the plus symbol (+).

```
messageText + firstName
```

Instead of just **firstName** between the round brackets of `MessageBox.Show()`, we can add the **messageText** variable and the plus symbol:

```
MessageBox.Show( messageText + firstName );
```

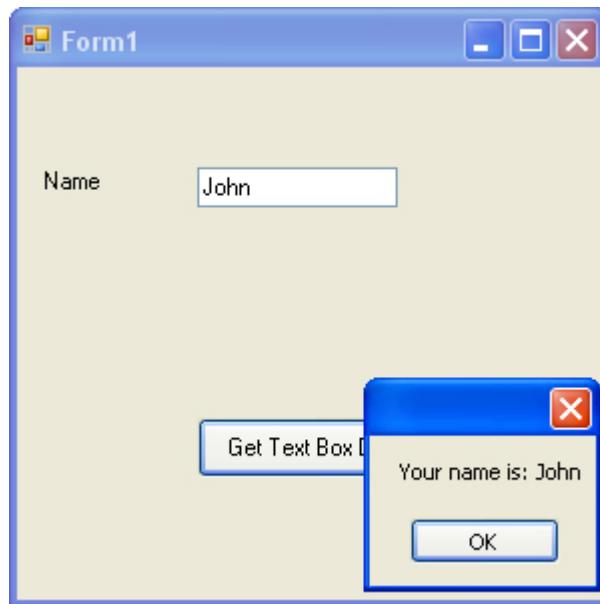
Amend your `MessageBox` line so it’s the same as the one above. Here’s the coding window:

```
private void btnStrings_Click(object sender, EventArgs e)
{
    string firstName;
    string messageText;

    messageText = "Your name is: ";

    firstName = textBox1.Text;
    MessageBox.Show(messageText + firstName);
}
```

Run your programme. Type your first name into the text box, and then click your button. You should see something like this:



So we set up a variable to hold some direct text, and got the person's name from the text box. We stored this information in two different variables. To join the two together, we used the plus symbol. We then displayed the result in a message box.

But you can also do this:

```
MessageBox.Show( "Your name is: " + firstName );
```

Here, we're not storing the text in a variable called **messageText**. Instead, it's just direct text surrounded by double quotes. Notice, though, that we still use the plus symbol to join the two together.

Using other controls to display text

You don't have to use a message box to display the result. You can use other controls, like a Label. Let's try it.

Add a new Label to your form. Use the Properties Window to set the following properties for your new Label:

Name: TextMessage
Location: 87, 126
Text: Message Area

Return to your coding window, and add two forward slashes to the start of your **MessageBox.Show()** line. The line should turn green, as in the following image:

```
private void btnStrings_Click(object sender, EventArgs e)
{
    string firstName;
    string messageText;

    messageText = "Your name is: ";

    firstName = textBox1.Text;

    //MessageBox.Show(messageText + firstName);

}
```

The reason it turns green is that two forward slashes are the characters you use to add a comment. C# then ignores these lines when running the programme. Comments are a very useful way to remind yourself what the programme does, or what a particular part of your code is for. Here's our coding window with some comments added:

```
=====//
// THIS BUTTON GETS INFORMATION FROM A TEXT BOX
=====//
private void btnStrings_Click(object sender, EventArgs e)
{
    string firstName;
    string messageText;

    messageText = "Your name is: ";

    =====//
// GET THE TEXT FROM THE TEXT BOX
=====//
    firstName = textBox1.Text;

    //MessageBox.Show(messageText + firstName);

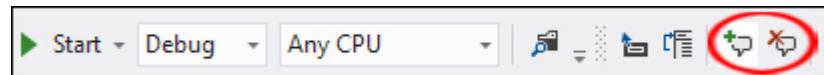
}
```

You can also use the menu bar, or the toolbar, to add comments. Highlight any line of text in your code. From the menu bar at the top of Visual Studio, select **Edit > Advanced > Comment Selection**. Two forward slashes will be added to

the start of the line. You can quickly add or remove comments by using the toolbar. Locate the following icons on the toolbars at the top of Visual Studio:



In version 2012, the comment icons look like this:



The comment icons are circled, in the images above. The first icon adds a comment, and the second one removes a comment. (If you can't see the above icons anywhere on your toolbars, click **View > Toolbars > Text Editor**.)

Now that you have commented out the MessageBox line, it won't get executed when your code runs. Instead, add the following line to the end of your code:

```
TextMessage.Text = messageText + firstName;
```

Your coding window should then look like this:

```
private void btnStrings_Click(object sender, EventArgs e)
{
    string firstName;
    string messageText;

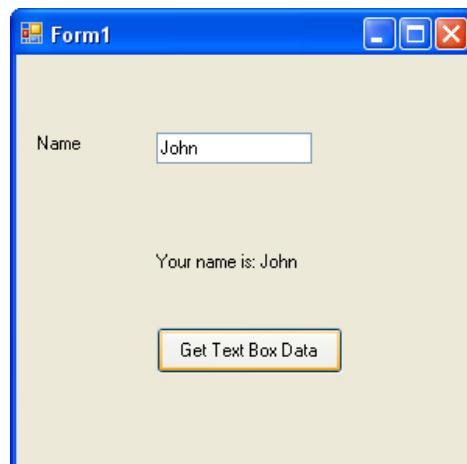
    messageText = "Your name is: ";

    firstName = textBox1.Text;

    //MessageBox.Show(messageText + firstName);

    TextMessage.Text = messageText + firstName;
}
```

Run your programme again. Type your name in the text box, and then click your button. The message should now appear on your label, instead of in a Message Box:



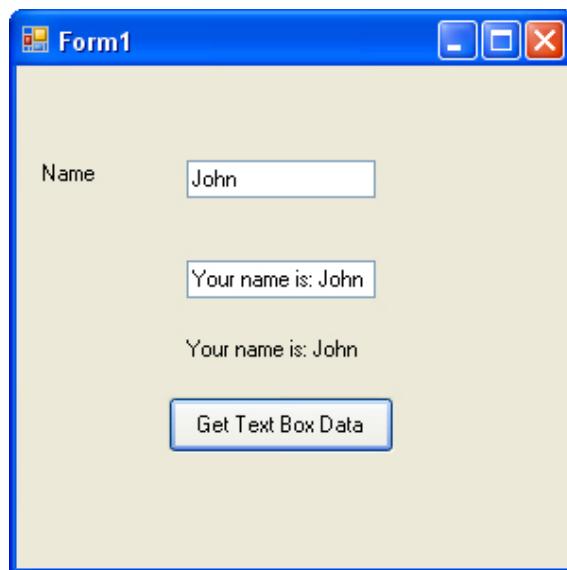
The reason it does so is because you're now setting the Text property of the Label with code. Previously, you changed the Label's Text Property from the Properties Window. The name of our label is **TextMessage**. To the right of the equals sign, we have the same code that was in between the round brackets of the Show() method of the MessageBox.

OK, time for an exercise.

Exercise

Add a second text box to your form. Display your message in the text box as well as on the label. So if your name is John, your second text box should have: "Your name is: John" in it after the button is clicked.

When you complete this exercise, your form should look like this, when the button is clicked:



We're now going to move away from string variables and on to number variables. The same principles you've just learnt still apply, though:

- Set up a variable, and give it a name
- Store something in the variable
- Use code to manipulate what you have stored

Variables - Numbers

There are a few different types of number variables. The ones you'll learn about now are **Integer**, **Double** and **Float**. First up, though, are Integers.

Close any solution you have open by clicking **File > Close Solution** from the menu bar at the top of Visual Studio. Start a new project by clicking **File > New Project**. From the New Project dialogue box, select **Windows Forms Application** from the available templates. Type a Name for your project. Call it **Numbers**.

Click OK, and you'll have a new form to work with.

Integers

An integer is a whole number. It's the 6 of 6.5, for example. In programming, you'll work with integers a lot. But they are just variables that you store in memory and want to manipulate.

Add a button to your form, and set the following properties for it in the Properties Window:

Name: btnIntegers
Text: Integers
Location: 110, 20

Now double click your button to get at the code:

```
namespace Numbers
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

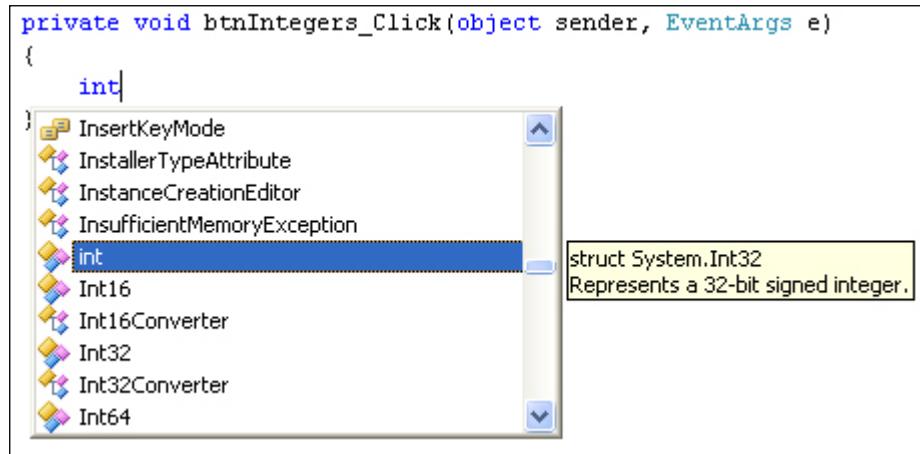
        private void btnIntegers_Click(object sender, EventArgs e)
        {
        }
    }
}
```

In the previous section, you saw that to set up a string variable you just did this:

string myText;

You set up an integer variable in the same way. Except, instead of typing the word **string**, you type the word **int** (short for integer).

So, in between the curly brackets of your button code, type **int**. You should see the word turn blue, and the IntelliSense list appear:



Either press the enter key on your keyboard, or just hit the spacebar. Then type a name for your new variable. Call it **myInteger**. Add the semi-colon at the end of your line of code, and hit the enter key. Your coding window will then look like this:

```
private void btnIntegers_Click(object sender, EventArgs e)
{
    int myInteger;
}
```

Notice the text in the yellow box, in the image one up from the one above. It says:

Represents a 32-bit signed integer

A signed integer is one that can have negative values, like -5, -6, etc. (The opposite, no negative numbers, is called an unsigned integer.) The 32-bit part is referring to the range of numbers that an integer can hold. The maximum value that you can store in an integer is: 2,147,483,648. The minimum value is the same, but with a minus sign on the front: -2,147,483,648

To store an integer number in your variable, you do the same as you did for string: type the name of your variable, then an equals sign (=), then the number you want to store. So add this line to your code (don't forget the semi-colon on the end):

myInteger = 25;

Your coding window should look like this:

```
private void btnIntegers_Click(object sender, EventArgs e)
{
    int myInteger;

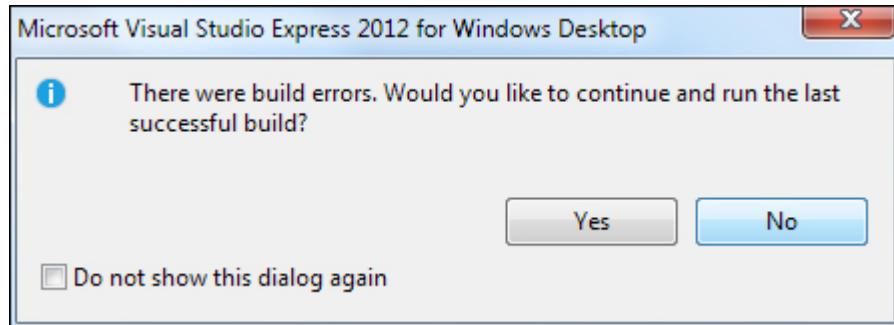
    myInteger = 25;
}
```

So we've set up an integer variable called **myInteger**. On the second line, we're storing a value of 25 inside of the variable.

We'll use a message box to display the result when the button is clicked. So add this line of code for line three:

```
MessageBox.Show(myInteger);
```

Now try to run your code. You'll get the following error message:



You should see a blue wiggly line under your MessageBox code:

```
private void btnIntegers_Click(object sender, EventArgs e)
{
    int myInteger;

    myInteger = 25;
    MessageBox.Show(myInteger);
}
```

Hold your mouse over `myInteger`, between the round brackets of `Show()`. You should see the following yellow box:

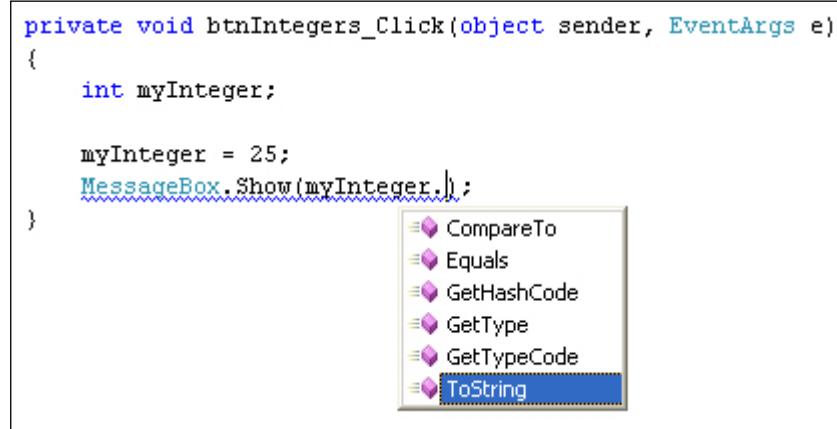
```
private void btnIntegers_Click(object sender, EventArgs e)
{
    int myInteger;

    myInteger = 25;
    MessageBox.Show(myInteger);
}          Argument '1': cannot convert from 'int' to 'string'
```

The error is: "Cannot convert from int to string". The reason you get this error is because **myInteger** holds a number. But the MessageBox only displays text. C#

does not convert the number to text for you. It doesn't do this because C# is a programming language known as “strongly typed”. What this means is that you have to declare the type of variable you are using (string, integer, double). C# will then check to make sure that there are no numbers trying to pass themselves off as strings, or any text trying to pass itself off as a number. In our code above, we're trying to pass **myInteger** off as a string. And C# has spotted it!

What you have to do is to convert one type of variable to another. You can convert a number into a string quite easily. Type a full stop (period) after the “r” of **myInteger**. You'll see the IntelliSense list appear:



```

private void btnIntegers_Click(object sender, EventArgs e)
{
    int myInteger;

    myInteger = 25;
    MessageBox.Show(myInteger.);
}

```

The screenshot shows a code editor with the following snippet:

```

private void btnIntegers_Click(object sender, EventArgs e)
{
    int myInteger;

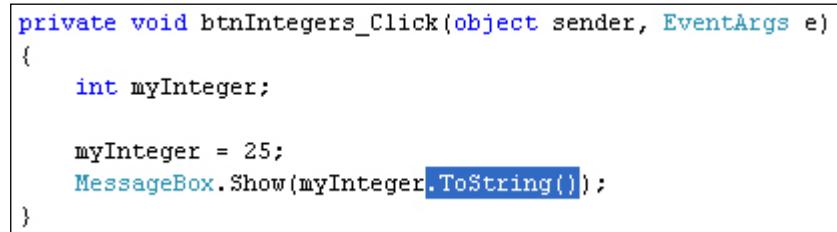
    myInteger = 25;
    MessageBox.Show(myInteger.);
}

```

A tooltip window is open over the period character after "myInteger.". The tooltip lists several methods starting with "ToString":

- ≡ CompareTo
- ≡ Equals
- ≡ GetHashCode
- ≡ GetType
- ≡ GetTypeCode
- ≡ **ToString**

Select **ToString** from the list. Because **ToString** is a method, you need to type a pair of round brackets after the “g” of **ToString**. Your code will then look like this (we've highlighted the new addition):



```

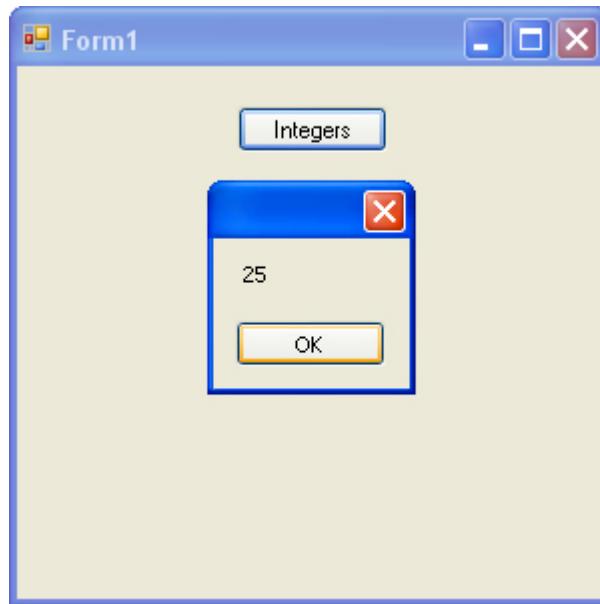
private void btnIntegers_Click(object sender, EventArgs e)
{
    int myInteger;

    myInteger = 25;
    MessageBox.Show(myInteger.ToString());
}

```

The **ToString** method, as its name suggests, converts something to a string of text. The thing we are converting is an integer.

Start your programme again. Because you've converted an integer to a string, you should find that it runs OK now. Click your button and you should see the message box appear:



Doubles and Float

Integers, as was mentioned, are whole numbers. They can't store the point something, like .7, .42, and .007. If you need to store numbers that are not whole numbers, you need a different type of variable. You can use the **Double** type, or the **float** type. You set these types of variables up in exactly the same way: instead of using the word **int**, you type **double**, or **float**. Like this:

```
float myFloat;  
double myDouble;
```

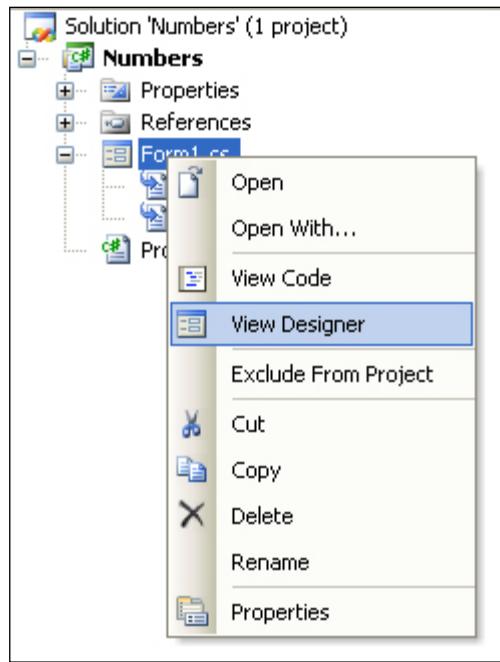
(Float is short for “floating point”, and just means a number with a point something on the end.)

The difference between the two is in the size of the numbers that they can hold. For float, you can have up to 7 digits in your number. For doubles, you can have up to 16 digits. To be more precise, here's the official size:

| | |
|----------------|---|
| float: | 1.5×10^{-45} to 3.4×10^{38} |
| double: | 5.0×10^{-324} to 1.7×10^{308} |

Float is a 32-bit number and double is a 64-bit number.

To get some practice using floats and doubles, return to your form. If you can't see the **Form1.cs [Design]** tab at the top, right click **Form1.cs** in the Solution Explorer on the right hand side. (If you can't see the Solution Explorer, click **View > Solution Explorer** from the menu bar at the top.)



Add a new button to your form. Set the following properties for it in the Properties Window:

| | |
|------------------|----------|
| Name | btnFloat |
| Location: | 110, 75 |
| Text: | Float |

Double click your new button, and add the following line to the button code:

```
float myFloat;
```

Your coding window will then look like this:

```
private void btnIntegers_Click(object sender, EventArgs e)
{
    int myInteger;

    myInteger = 25;
    MessageBox.Show(myInteger.ToString());
}

private void btnFloat_Click(object sender, EventArgs e)
{
    float myFloat;
}
```

To store something inside of your new variable, add the following line:

```
myFloat = 0.42F;
```

The capital letter F on the end means Float. You can leave it off, but C# then treats it like a double. Because you've set the variable up as a float, you'll get errors if you try to assign a double to a float variable.

Add a third line of code to display your floating point number in a message box:

```
MessageBox.Show( myFloat.ToString() );
```

Again, we have to use `ToString()` in order to convert the number to a string of text, so that the message box can display it.

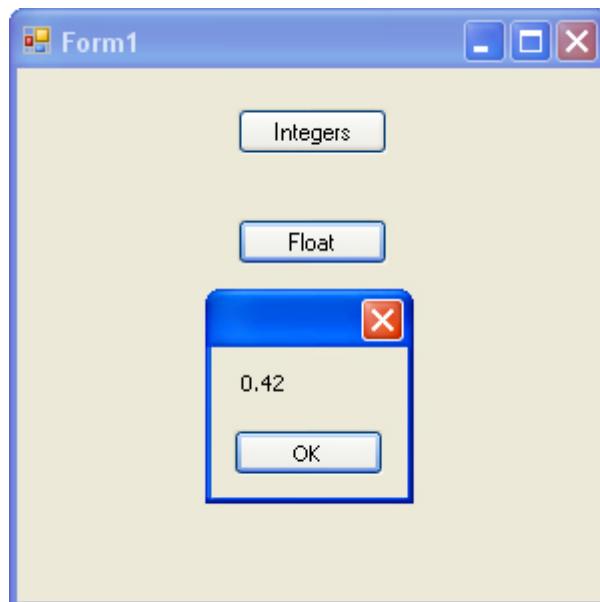
But your coding window should look like ours below:

```
private void btnFloat_Click(object sender, EventArgs e)
{
    float myFloat;

    myFloat = 0.42F;

    MessageBox.Show( myFloat.ToString() );
}
```

Run your programme and click your Float button. You should see a form like this:



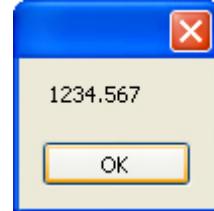
Halt the programme and return to your coding window. Now delete the capital letter F from 0.42. The line will then be:

```
myFloat = 0.42;
```

Try to run your programme again. You'll get an error message, and a blue wiggly line under your code. Because you've missed the F out, C# has defaulted to using a double value for your number. A float variable can't hold a double value, confirming that C# is a strongly typed language. (The opposite is a weakly-typed

language. PHP and JavaScript are examples of weakly typed languages – you can store any kind of values in the variables you set up.)

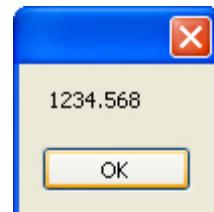
Another thing to be careful of when using float variables is rounding up or down. As an example, change the number from 0.42F to 1234.567F. Now run your programme, and click your float button. The message box will be this:



Halt the programme and return to your code. Now add an 8 before the F and after the 7, so that your line of code reads:

```
myFloat = 1234.5678F;
```

Now run your programme again. When you click the button, your message box will be this:



It's missed the 7 out! The reason for this is that float variables can only hold 7 numbers in total. If there's more than this, C# will round up or down. A number that ends in 5 or more will be rounded up. A number ends in 5 or less will be rounded down:

| | |
|------------------|---|
| 1234.5678 | (eight numbers ending in 8 – round up) |
| 1234.5674 | (eight numbers ending in 4 – round down) |

The number of digits that a variable can hold is known as precision. For float variable, C# is precise to 7 digits: anything more and the number is rounded off.

Double values

Add another button to your form, and set the following properties for it in the Properties Window:

| | |
|------------------|------------------|
| Name | btnDouble |
| Location: | 110, 130 |
| Text: | Double |

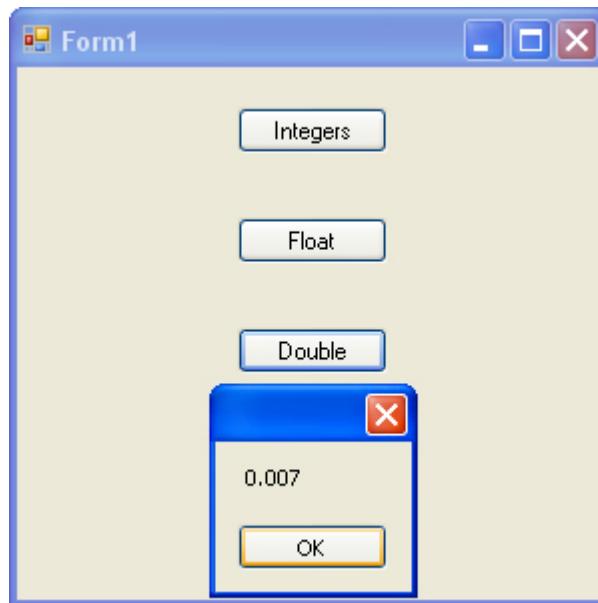
Double click your new button to get at the code. Add the following three lines to your button code:

```
double myDouble;  
  
myDouble = 0.007;  
  
MessageBox.Show(myDouble.ToString());
```

Your coding window should now look like this:

```
private void btnDouble_Click(object sender, EventArgs e)  
{  
    double myDouble;  
  
    myDouble = 0.007;  
  
    MessageBox.Show(myDouble.ToString());  
}
```

Run your programme and click your new button. You should see this:



You also need to be careful of precision when using double variable types. The double type can hold up to 16 digits.

Halt your programme and return to the coding window. Change this line:

myDouble = 0.007;

to this:

myDouble = 12345678.1234567;

Run your programme and click your double button. The message box correctly displays the number. Add another number on the end, though, and C# will again round up or down. The moral is, if you want accuracy, careful of rounding!

Simple addition

We'll now use variables to do some adding up. After you have learned how to add up with the three number variable types, we can move on to subtraction, multiplication, and division.

Start a new project for this. So click **File > Close Solution** from the menu bar at the top of Visual Studio. Then click **File > New Project**. Type **arithmetic** as the Name of your new **Windows Forms Application** project. Click OK to create the new project.

Add a button to your new form, and set the following properties for it in the Properties Window:

| | |
|--------------|---------------|
| Name: | btnAdd |
| Size: | 100, 30 |
| Text: | Integer - Add |

Move the button to the top of your form. Then double click it to get at the coding window. Set up the following three integer variables in your button code:

```
int firstNumber;  
int secondNumber;  
int integerAnswer;
```

Your coding window should look like ours below:

```
private void btnAdd_Click(object sender, EventArgs e)  
{  
    int firstNumber;  
    int secondNumber;  
    int integerAnswer;  
}
```

We now need to put something into these variables. We'll store 10 in the first number, and 32 in the second number. So add these two lines to your code:

```
firstNumber = 10;  
secondNumber = 32;
```

Your coding window will then look like this:

```
private void btnAdd_Click(object sender, EventArgs e)
{
    int firstNumber;
    int secondNumber;
    int integerAnswer;

    firstNumber = 10;
    secondNumber = 32;
}
```

So the numbers we want to store in the variables go on the right hand side of the equals sign; the variable names go on the left hand side of the equals sign. This assigns the numbers to the variables – puts them into storage.

We now want to add the first number to the second number. The result will be stored in the variable we've called **integerAnswer**. Fortunately, C# uses the plus symbol (+) to add up. So it's fairly simple. Add this line to your code:

```
integerAnswer = firstNumber + secondNumber;
```

And here's the coding window:

```
private void btnAdd_Click(object sender, EventArgs e)
{
    int firstNumber;
    int secondNumber;
    int integerAnswer;

    firstNumber = 10;
    secondNumber = 32;

    integerAnswer = firstNumber + secondNumber;
}
```

We've already stored the number 10 in the variable called **firstNumber**. We've stored 32 in the variable **secondNumber**. So we can use the variable names to add up. The two variables are separated by the plus symbol. This is enough to tell C# to add up the values in the two variables. The result of the addition then gets stored to the left of the equals sign, in the variable called **integerAnswer**. Think of it like this:

```
private void btnAdd_Click(object sender, EventArgs e)
{
    int firstNumber;
    int secondNumber;
    int integerAnswer;

    firstNumber = 10;
    secondNumber = 32;

    integerAnswer = firstNumber + secondNumber;
}
```

Calculate this sum first

```
private void btnAdd_Click(object sender, EventArgs e)
{
    int firstNumber;
    int secondNumber;
    int integerAnswer;

    firstNumber = 10;
    secondNumber = 32;

    integerAnswer = firstNumber + secondNumber;
}
```

Store the answer here

To see if all this works or not, add a message box as the final line of code:

```
MessageBox.Show( integerAnswer.ToString() );
```

We're just placing the **integerAnswer** variable between the round brackets of **Show()**. Because it's a number, we've had to use **ToString()** to convert the number to text. Here's what your coding window should look like now:

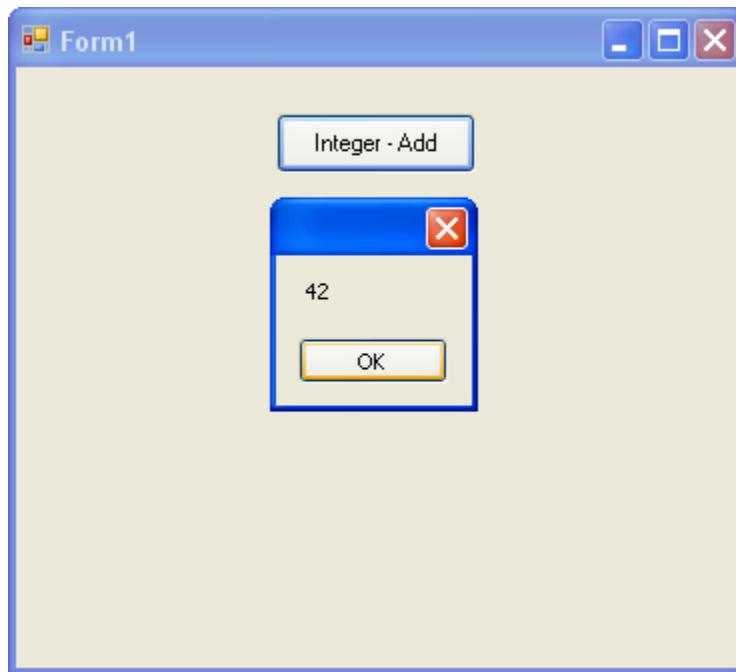
```
private void btnAdd_Click(object sender, EventArgs e)
{
    int firstNumber;
    int secondNumber;
    int integerAnswer;

    firstNumber = 10;
    secondNumber = 32;

    integerAnswer = firstNumber + secondNumber;

    MessageBox.Show( integerAnswer.ToString() );
}
```

And here's the form when the button is clicked:



You don't have to store numbers in variables if you want to calculate things. You can just add up the numbers themselves. Like this:

```
integerAnswer = 10 + 32;
```

And even this:

```
integerAnswer = firstNumber + 32;
```

So you can add up just using numbers, or you can mix variable names with numbers. As long as C# knows that there's a number in your variable, and that it's the right type, the addition will work.

You can use more than two variables, or more than two numbers. So you can do this:

```
integerAnswer = firstNumber + secondNumber + thirdNumber;
```

or this:

```
integerAnswer = firstNumber + secondNumber + 32;
```

And this:

```
integerAnswer = firstNumber + 10 + 32;
```

The results is the same: C# adds up whatever you have on the right hand side of the equals sign, and then stores the answer on the left hand side.

Adding up with float Variables

You add with float variables in exactly the same way – with the plus symbol. You can even mix integer variables with float variables. But you have to take care!

Add another button to your form, and set the following properties for it in the Properties Window:

| | |
|--------------|--------------|
| Name: | btnAddFloats |
| Size: | 100, 30 |
| Text: | Float – Add |

Double click your button to get at the code. Set up the following variables:

```
float firstNumber;
float secondNumber;
float floatAnswer;
```

And here's the coding window:

```
private void btnAddFloats_Click(object sender, EventArgs e)
{
    float firstNumber;
    float secondNumber;
    float floatAnswer;
}
```

(Notice that we've used the same names for the first two variables. C# doesn't get confused, because they are in between the curly brackets of the button code. You can set up variables outside of the curly brackets. We'll do this when we come to code the calculator, at the end of this section. Then something called **scope** comes in to play.)

To place something in your new variables, add the following code:

```
firstNumber = 10.5F;  
secondNumber = 32.5F;  
  
floatAnswer = firstNumber + secondNumber;
```

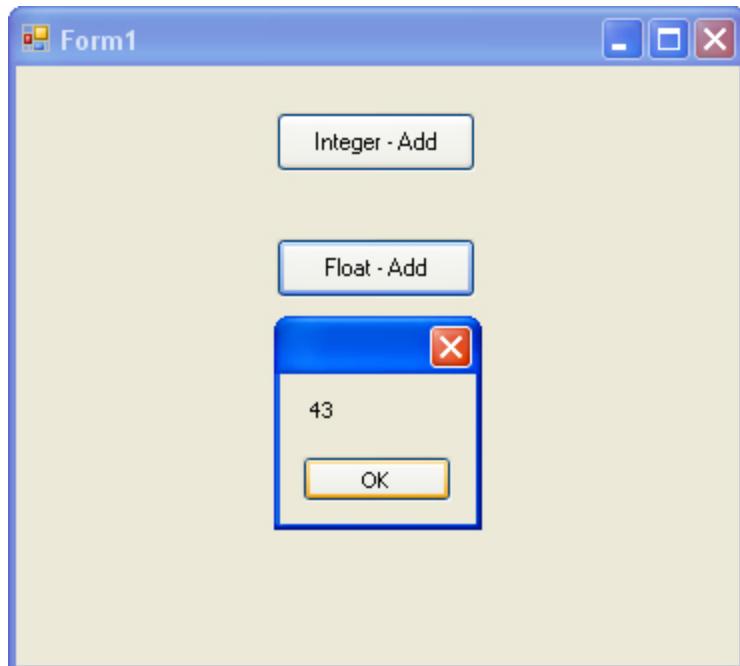
Finally, add you message box line:

```
MessageBox.Show( floatAnswer.ToString() );
```

The coding window should look like this:

```
private void btnAddFloats_Click(object sender, EventArgs e)  
{  
    float firstNumber;  
    float secondNumber;  
    float floatAnswer;  
  
    firstNumber = 10.5F;  
    secondNumber = 32.5F;  
  
    floatAnswer = firstNumber + secondNumber;  
  
    MessageBox.Show( floatAnswer.ToString() );  
}
```

Run your form and click your new button. You should see this:



So $10.5 + 32.5$ equals 43. Halt your form by clicking the red X, and return to your coding window.

As was mentioned, you can add float and integer values together. But you need to take care. Try this:

Add the following variable to your code:

```
int integerAnswer;
```

And then change this line:

```
floatAnswer = firstNumber + secondNumber;
```

to this:

```
integerAnswer = firstNumber + secondNumber;
```

So it's just the name of the variable before the equals sign that needs to be changed.

Amend your message box line from this:

```
MessageBox.Show( floatAnswer.ToString() );
```

to this:

```
MessageBox.Show( integerAnswer.ToString() );
```

Your coding window will then look like this:

```
private void btnAddFloats_Click(object sender, EventArgs e)
{
    float firstNumber;
    float secondNumber;
    float floatAnswer;
    int integerAnswer;

    firstNumber = 10.5F;
    secondNumber = 32.5F;

    integerAnswer = firstNumber + secondNumber;

    MessageBox.Show(integerAnswer.ToString());
}
```

Try to run your code. The programme won't execute, and you'll have a blue wiggly line:

```
private void btnAddFloats_Click(object sender, EventArgs e)
{
    float firstNumber;
    float secondNumber;
    float floatAnswer;
    int integerAnswer;

    firstNumber = 10.5F;
    secondNumber = 32.5F;

    integerAnswer = firstNumber + secondNumber;

    MessageBox.Show(integerAnswer.ToString());
}
```

Hold your mouse over the blue wiggly line and you'll see an explanation of the error:

Cannot implicitly convert type 'float' to 'int'. An explicit conversion exists (are you missing a cast?)

Not much help, if you're a beginner! But what it's telling you is that the first number and the second number are float variables. The answer to the addition was also a float. However, you were trying to store the answer in an integer variable. C# won't let you store float values in an integer. The error message is saying that you need to convert them first.

You can indeed convert float values to integers. You do it like this:

```
integerAnswer = (int)firstNumber + (int)secondNumber;
```

So you type the word **int** between a pair of round brackets. This goes before the number you want to convert. It does mean that the point something on the end will get chopped off, though. So 10.5 becomes 10, and 32.5 becomes 32. Not good for accuracy, but at least the programme will run!

Try it out, and you should see an answer of 42 when you click your button.

So the moral is this: If you're expecting an answer that ends in point something, use a float variable (or a double).

(You may have a green wiggly line under **float floatAnswer**. This is because you're not storing anything in this variable. Don't worry about it!)

Note that the other way round is not a problem – you can store an integer in a float value. Have a look at this slight change to the code:

```
private void btnAddFloats_Click(object sender, EventArgs e)
{
    float firstNumber;
    float secondNumber;
    float floatAnswer;
    int integerAnswer = 20;

    firstNumber = 10.5F;
    secondNumber = 32.5F;

    floatAnswer = firstNumber + secondNumber + integerAnswer;

    MessageBox.Show(floatAnswer.ToString());
}
```

First, notice the new way we are storing the number 20 into the integer variable called **integerAnswer**:

```
int integerAnswer = 20;
```

Instead of two lines, we've just used one. This is fine, in C#. But you're doing two things on the same line: setting up the variable, and placing a value in it.

The second thing to notice is that we are adding up two float values (**firstNumber** and **secondNumber**) and an integer (**integerAnswer**). We're then storing the answer into a float variable (**floatAnswer**). Try it out and you'll find that the code runs fine.

If we change this line:

```
firstNumber = 10.5F;
```

to this:

```
firstNumber = 10;
```

then, again, the programme will run fine. In other words, you can store an integer in a float variable, but you can't store a float value in an integer variable.

Hopefully, that wasn't too confusing!

We'll move on to subtraction, now. But if you want to use a double variable instead of a float variable the same things apply – be careful of what you are trying to store, and where!

Subtraction

Subtraction is fairly straightforward in C#. To subtract one number from another, you used the minus symbol (-).

Add another button to your form. Set the following properties for it in the Properties Window:

| | |
|--------------|-------------|
| Name: | btnSubtract |
| Size: | 100, 30 |
| Text: | Subtract |

Double click the button to get at the code. Add the following three lines of code:

```
int answerSubtract;

answerSubtract = 50 - 25;

MessageBox.Show( answerSubtract.ToString() );
```

Your coding window will then look like this:

```
private void btnSubtract_Click(object sender, EventArgs e)
{
    int answerSubtract;

    answerSubtract = 50 - 25;

    MessageBox.Show( answerSubtract.ToString() );
}
```

So we've set up an integer variable called **answerSubtract**. On the second line, we're using the minus symbol to subtract 25 from 50. When C# works out the answer to $50 - 25$, it will place the answer to the left of the equals sign, in the **answerSubtract** variable. On the final line, we're displaying the answer in a message box.

Run your code, and make sure it works. The answer you should see in the message box is, of course, 25. Stop your programme and return to the coding window. Now change the 25 to 25.5.

answerSubtract = 50 - 25.5;

Try to run your programme and you'll get the blue wiggly line, meaning there's an error in your code. The reason is the same as for addition: we're trying to place a float number into an integer variable (the answer will be 24.5, this time). Just because the math symbol has changed doesn't mean we can disobey the C# rules!

Change it back to 25, and the code will run fine.

As with addition, you can subtract more than one number. Change your line to this:

```
answerSubtract = 50 - 25 - 10 - 2;
```

When you run your programme, you should see 13 in your message box, the answer to the subtraction.

You can also use variable names in your subtraction. Add the following integer variable to your code:

```
int numberOne = 12;
```

Then change the second line to this:

```
answerSubtract = 50 - numberOne;
```

Here's what your coding window should look like:

```
private void btnSubtract_Click(object sender, EventArgs e)
{
    int answerSubtract;
    int numberOne = 12;

    answerSubtract = 50 - numberOne;

    MessageBox.Show( answerSubtract.ToString() );
}
```

What we're doing here is setting up an integer variable called **numberOne**. We're then placing a value of 12 inside of the variable – all on the same line. For the second line, we're subtracting whatever is in the variable called numberOne from 50.

Run your programme and click your button. You should see an answer of 38 in your message box.

Exercise A

Set up another variable. Call it **numberTwo**. Place a value of 4 inside of this new variable. Subtract from 50 the value in numberOne and the value in numberTwo. When you run your code, the answer you get in the message box should be 34.

If you can't do an exercise in this book, click the hyperlink below the question. You'll then jump to the answer at the back of the book. Do the same to get back to where you were.

[Answer to Exercise A](#)

Mixing Subtraction and Addition

You can mix subtraction and addition. The process is quite straightforward. What we'll do next is to add two numbers together, and then subtract a third number from the total.

Add another button to your form. Set the following properties for it:

Name: btnMixed
Size: 100, 30
Text Add and Subtract

(If you need to make your Form bigger, click on the form to select it. Then change the Size property in the Properties Window.)

Double click your button to get at the code. We'll need four integer variables for this. So set up the following:

```
int firstNumber;
int secondNumber;
int thirdNumber;
int answer;
```

To place values in the variables, add the following three lines:

```
firstNumber = 100;
secondNumber = 75;
thirdNumber = 50;
```

Your coding window will then look like this:

```
private void btnMixed_Click(object sender, EventArgs e)
{
    int firstNumber;
    int secondNumber;
    int thirdNumber;
    int answer;

    firstNumber = 100;
    secondNumber = 75;
    thirdNumber = 50;
}
```

To add the first number to the second number, and then place the result in the variable we've called **answer**, add the following line to your code:

```
answer = firstNumber + secondNumber;
```

Display the answer in a message box by adding this line:

```
MessageBox.Show( answer.ToString( ) );
```

When you run the programme and click your button, you should see the message box display an answer of 175.

Stop the programme and return to your code.

We now want to subtract the third number from the first two. So change this line:

```
answer = firstNumber + secondNumber;
```

to this:

```
answer = firstNumber + secondNumber - thirdNumber;
```

When C# sees all these variables after the equals sign, it will try to calculate using the numbers you've stored in the variables. Usually, it will calculate from left to right. So this bit gets done first:

firstNumber + secondNumber

When C# finishes adding the first two numbers, it will then deduct the value in **thirdNumber**. The answer is then stored in the variable to the left of the equals sign.

Run your programme. When the button is clicked, the message box will display an answer of 125.

Operator Precedence

The symbols you have been using (+ and -) are known as Operators. Operator Precedence refers to the order in which things are calculated. C# sees the plus (+) and the minus (-) as being of equal weight, so it does the calculating from left to right. But this "left to right" calculating can cause you problems. Change the plus into a minus in your code, and the minus into a plus. So change it to this:

```
answer = firstNumber - secondNumber + thirdNumber;
```

Run your code. When the button is clicked, an answer of 75 will appear in the message box. This is "left to right" calculation. What you wanted here was:

firstNumber – secondNumber

Then when the answer to that is found, add the **thirdNumber**. So the sum is this: 100 – 75, which equals 25. Then 25 + 50, which equals 75.

But what if you didn't mean that? What if you wanted `firstNumber` minus the answer to `secondNumber + thirdNumber`? In case that's not clear, some brackets may help clear things up. Here's the two ways of looking at our calculation:

$$(\text{firstNumber} - \text{secondNumber}) + \text{thirdNumber}$$

$$\text{firstNumber} - (\text{secondNumber} + \text{thirdNumber})$$

In maths, brackets are a way to clear up your calculations. In the first one, whatever is between the round brackets is calculated first. The total of the sum in brackets is then added to `thirdNumber`. In the second one, it's the other way around: `secondNumber` is first added to `thirdNumber`. You then deduct from this total the value of `firstNumber`.

You can use brackets in programming, as well. Add the following brackets to your code:

```
answer = firstNumber - (secondNumber + thirdNumber);
```

Your coding window should then look like this:

```
private void btnMixed_Click(object sender, EventArgs e)
{
    int firstNumber;
    int secondNumber;
    int thirdNumber;
    int answer;

    firstNumber = 100;
    secondNumber = 75;
    thirdNumber = 50;

    answer = firstNumber - (secondNumber + thirdNumber);

    MessageBox.Show( answer.ToString() );
}
```

When you run your programme and click the button, this time the answer is minus 25. Previously, the answer was 75! The reason the answer is different is because you've used brackets. C# sees the brackets and tackles this problem first. The next thing it does is to deduct `firstNumber` from the total. Without the brackets, it simply calculates from left to right.

Exercise

Try the following:

```
answer = ( firstNumber - secondNumber ) + thirdNumber;
```

What answer do you expect to get?

Multiplication and Division

To multiply and divide, the following symbols are used in programming:

| | |
|---|----------|
| * | Multiply |
| / | Divide |

Change your code to this:

```
answer = ( firstNumber + secondNumber ) * thirdNumber;
```

Because of the brackets, the first thing that C# does is to add the value in **firstNumber** to the value in **secondNumber**. The total is then multiplied (*) by the value in **thirdNumber**. With the values we currently have in the variables, the sum is this:

```
answer = ( 100 + 75 ) * 50
```

Run your programme and click your button. The answer you should get is 8750. Return to the coding window. Now remove the two round brackets. Your coding window will then be this:

```
private void btnMixed_Click(object sender, EventArgs e)
{
    int firstNumber;
    int secondNumber;
    int thirdNumber;
    int answer;

    firstNumber = 100;
    secondNumber = 75;
    thirdNumber = 50;

    answer = firstNumber + secondNumber * thirdNumber;

    MessageBox.Show( answer.ToString() );
}
```

Run your programme again. Click the button. This time, the answer is 3850! The reason it's different is because of Operator Precedence. With the brackets, you forced C# to calculate the addition first. Without the brackets, C# no longer calculates from left to right. So it's not doing this:

(100 + 75) * 50

C# sees multiplication as having priority over addition and subtraction. So it does the multiplying first. It does this:

100 + (75 * 50)

The two give you a totally different answer.

The same is true of division. Try this. Amend your line of code to the following:

```
answer = ( firstNumber + secondNumber ) / thirdNumber;
```

Run the programme and the answer will be 3. (The real answer to $(100 + 75) / 50$ is, of course, 3.5. But because we're using integer variables and not floats, the point something at the end gets chopped off.)

So we're using the divide symbol now (/), instead of the multiplication symbol (*). The addition is done first, because we have a pair of round brackets. The total of the addition is then divided by the value in **thirdNumber**.

Return to the code, and change the line to this:

```
answer = firstNumber + secondNumber / thirdNumber;
```

So just remove the round brackets. Run your programme again, and you'll find that the answer in the message box is now 101. (It would have been 101.5, if we had used float variables instead of integers.)

If you now replace the plus symbol (+) above with a multiplication symbol (*), C# switches back to "left to right" calculation. This is because it sees division and multiplication as having equal weight. The answer you'll get without brackets is 150.

Exercise B

Try the following two lines. First this one:

```
answer = (firstNumber * secondNumber) / thirdNumber;
```

And now this one:

```
answer = firstNumber * (secondNumber / thirdNumber);
```

What answer do you get with the round brackets in different places? Can you understand why? If not, go back over this section.

[Answer to Exercise B](#)

Getting numbers from text boxes

We're going to change tack slightly, here. What we'll do is show you how to get numbers from text boxes, and then use these numbers in your code. You'll need to be able to do this for your calculator.

Start a new project for this one by clicking **File > New Project** from the menu bar at the top of Visual Studio.

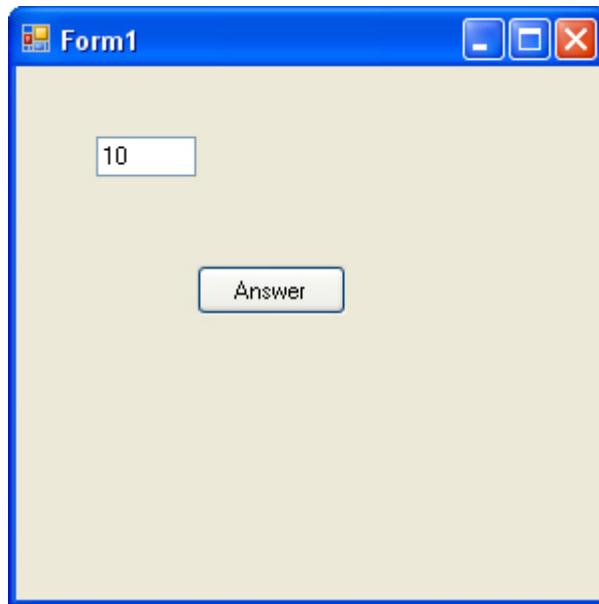
Add a text box and a button to your new form. Set the following Properties for the text box (the **tb** below stands for text box):

Name: tbFirstNumber
Size: 50, 20
Location: 40, 35
Text: 10

And set the following properties for your button:

Name: btnAnswer
Size: 75, 25
Location: 90, 90
Text: Answer

Your form will then look like this:



What we want to do is to get that number 10 from the text box and display it in a message box.

So double click your button to get at the coding window. Your cursor will be flashing inside of the button code. Set up two integer variables at the top of the button code:

```
int firstTextBoxNumber;  
int answer;
```

Your coding window should look like this:

```
private void btnAnswer_Click(object sender, EventArgs e)
{
    int firstTextBoxNumber;
    int answer;
}
```

To get at the number in the text box, we can use the **Text** property of text boxes. Here's the line of code to add:

```
firstTextBoxNumber = tbFirstNumber.Text;
```

This says, find a text box called **tbFirstNumber**. Access its **Text** property. When the **Text** property is retrieved, store it in the variable called **firstTextBoxNumber**.

To display the number in a message box, add this line:

```
MessageBox.Show( firstTextBoxNumber.ToString() );
```

Try to Run your code. You should find C# won't run it at all. It will give you the following error:

```
private void btnAnswer_Click(object sender, EventArgs e)
{
    int firstTextBoxNumber;
    int answer;

    firstTextBoxNumber = tbFirstNumber.Text;
    Cannot implicitly convert type 'string' to 'int'
    MessageBox.Show( firstTextBoxNumber.ToString() );
}
```

With text boxes, the thing that you get is, not surprisingly, text. However, we're trying to store the text from the text box into an integer variable. C# won't let you do this – whole numbers belong in integer variables, not text. The error message is telling you that C# can't do the conversion from text to numbers for you – you have to do it yourself!

So we need to convert the text from the text box into an integer. The way you do this is to use something called Parsing. Fortunately, this involves nothing more complex than typing the word "**Parse**". You can do different types of Parses. Because we need to convert the text into an integer, we need an Integer Parse. So change the line to this:

```
firstTextBoxNumber = int.Parse( tbFirstNumber.Text );
```

So you type **int**, then a full stop. From the IntelliSense menu, you can double click **Parse**. In between a pair of round brackets, you type the text you want to convert.

In our case, the text is coming from a text box. But it doesn't have to. You can do this:

```
firstTextBoxNumber = int.Parse("10");
```

In the code above, the number is in double quotes. Double quotes mean that it is text. Using `int.Parse()` means that it will be converted to a number that you can store in an integer variable.

Run your programme and you'll find that it works OK now. (You'll have a green wiggly line under answer, but that's just because we haven't used this variable yet.) Click your button and the number 10 will appear in the message box. Type a different number in your text box, and click the button again. The new number should appear in place of the old one.

You can also Parse other types of variable. Like this:

```
float firstTextBoxNumber;  
firstTextBoxNumber = float.Parse(tbFirstNumber.Text);
```

Or this:

```
double firstTextBoxNumber  
firstTextBoxNumber = double.Parse(tbFirstNumber.Text);
```

In the first one, we've set up a float variable. We've then used `float.Parse()` to convert the text from the text box, so that it can be stored in the float variable. We've done exactly the same thing in the second example, to convert the text into a double.

Things get more complicated if you accidentally try to store a double value in a float variable – your programme will crash! You need to try to catch things like this with code. (You'll see how to test for errors like this later in the book.)

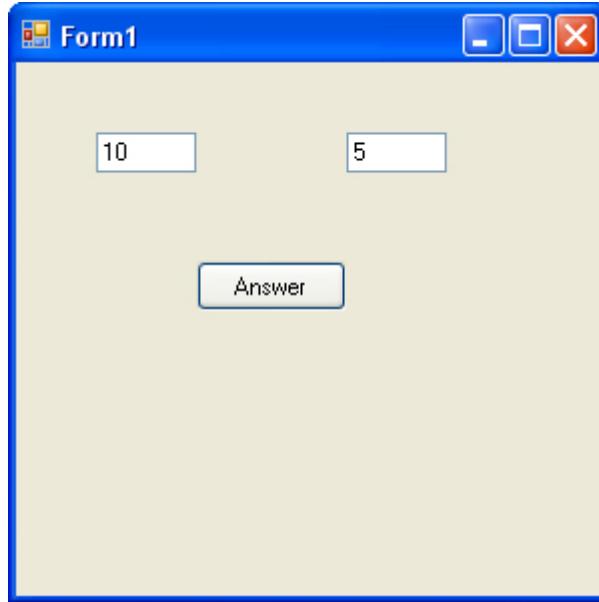
For now, let's move on.

OK, so we've grabbed a number from a text box and displayed it in a message box. What we'll do now is to add a second text box, get numbers from both, use our Math operators, and do some calculations with the two numbers we took from the text boxes. Sounds complex, but it isn't!

Add a second text box to your form. Set the following Properties for it in the Properties Window:

| | |
|------------------|----------------|
| Name: | tbSecondNumber |
| Size: | 50, 20 |
| Location: | 165, 35 |
| Text: | 5 |

Your form will then look like this:



Double click the button to get at your code. Now set up another integer variable to hold the second number from the new text box:

```
int secondTextBoxNumber;
```

To store the number from the text box in this new variable, add the following line:

```
secondTextBoxNumber = int.Parse( tbSecondNumber.Text );
```

This is the same as before – use `int.Parse` to convert the number from the text box into an integer variable. Then store the number in the new variable.

Let's add the two numbers up, first. We can use the **answer** variable for this. Here's the code to add:

```
answer = firstTextBoxNumber + secondTextBoxNumber;
```

So we're just using the plus symbol (+) to add up whatever is in the two variables. The numbers in the variables come from the two text boxes.

Amend your message box line to this:

```
MessageBox.Show( answer.ToString( ) );
```

All you need to do is to change the name of the variable that your are converting `ToString()`.

Your coding window should look like ours:

```
private void btnAnswer_Click(object sender, EventArgs e)
{
    int firstTextBoxNumber;
    int secondTextBoxNumber;
    int answer;

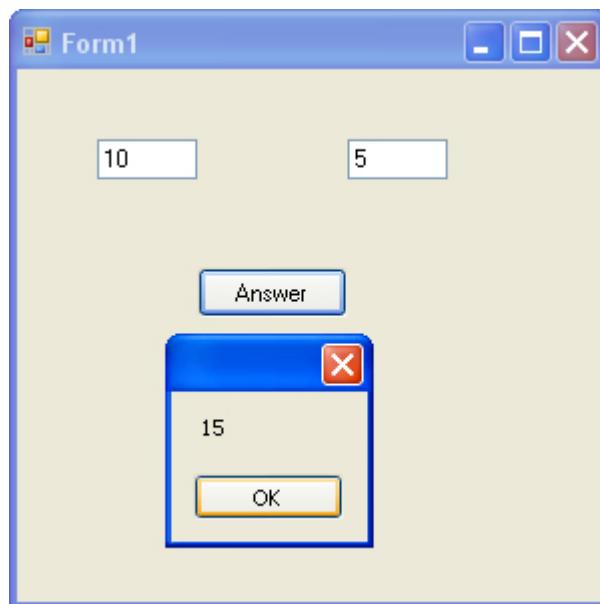
    firstTextBoxNumber = int.Parse(tbFirstNumber.Text);
    secondTextBoxNumber = int.Parse(tbSecondNumber.Text);

    answer = firstTextBoxNumber + secondTextBoxNumber;

    MessageBox.Show( answer.ToString() );

}
```

Run your programme, and then click the button. You should see the answer to the addition in your message box:



Change the numbers in the text boxes, and click your button again. When you've finished playing with your new form, click the red X to return to the code. Here's a few exercises for you to try.

Exercise

Use the textboxes on your form to calculate the following (you'll need to amend your code for three of them):

$$\begin{aligned} & 1845 + 285 \\ & 3450 - 285 \\ & 35 * 85 \\ & 5656 / 7 \end{aligned}$$

(The answers you should get are: 4703, 3165, 2975 and 808.)

Exercise

Add a new text box to your form. Set up an integer variable to store a third number. Get the third number from the text box and calculate the following:

$$(1845 + 2858) - 356$$

$$(3450 - 285) * 12$$

$$35 * (85 - 8)$$

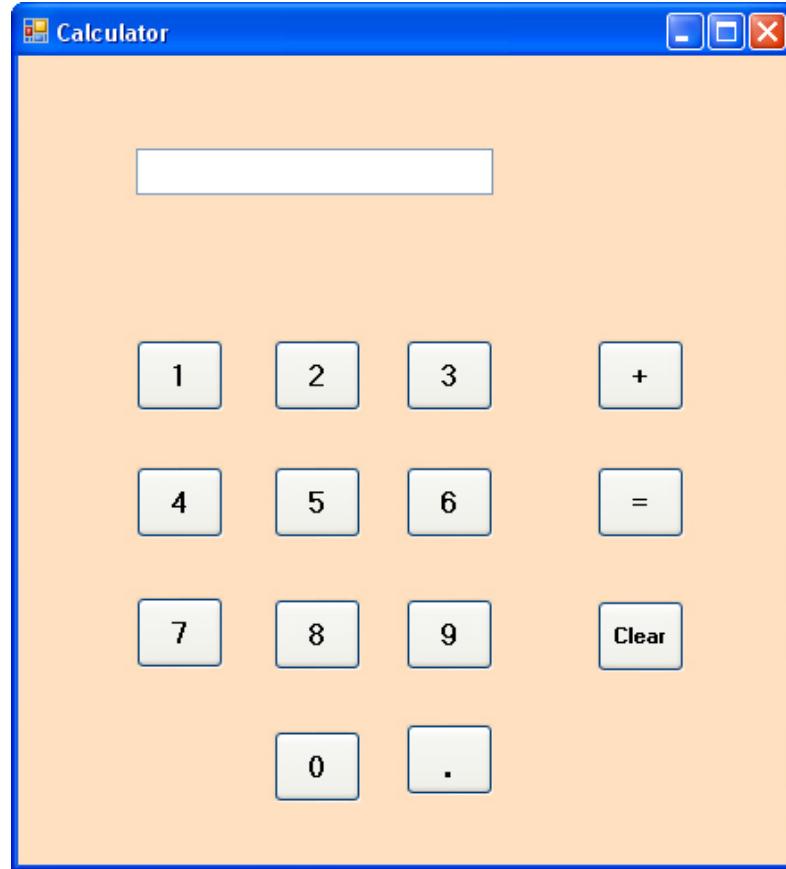
$$(5656 / 7) + 2156$$

(The answers you should get are: 4347, 37980, 2695 and 2964. You'll have to keep closing the form down. Then add round brackets, the operator symbols, and the new variable.)

Once you've completed the exercises, you can move on to tackling the next project – your very own calculator!

Calculator Project

You're now going to write your own very own C# calculator programme. We'll keep it simple at first, and the only thing it will be able to do is add up. After you understand how it all works, we'll make it divide, subtract and multiply. Version 1 of your calculator will look like this:

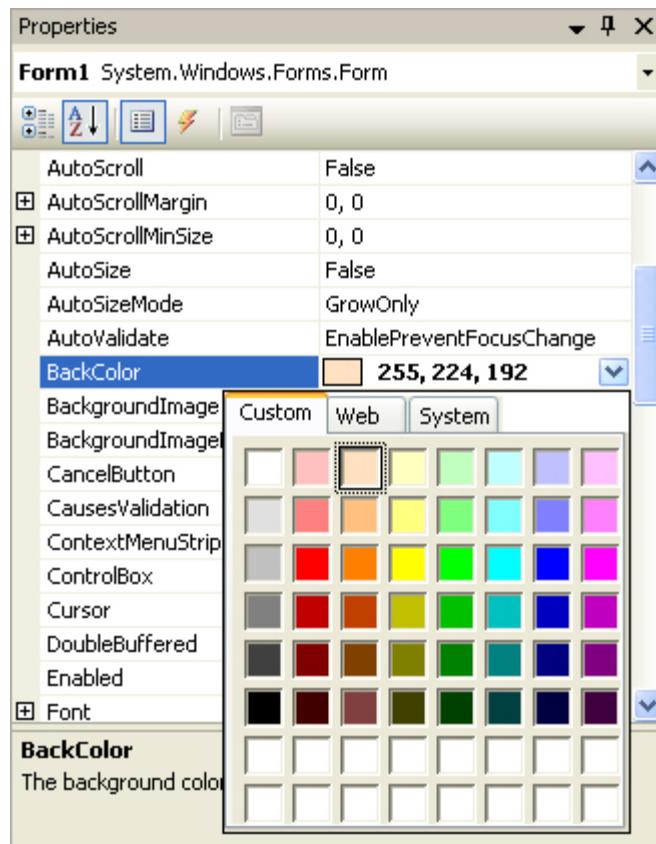


As you can see, it has a text box for the display of numbers, buttons for the numbers 0 to 9, a point symbol, plus and equals buttons, and a clear button.

So the first thing to do is to design your calculator. Start a new project by clicking **File > New project**. For your new form, set the following properties:

Size: 440, 487
Text: Calculator

To add a bit of colour to your calculator, you can change the **BackColour** property of the form, as in the image below:



We went for an orange colour, but feel free to chose any colour you like.

Now add a text box to your form and set the following properties for it:

Name: txtDisplay
Location: 66, 52
Size: 200, 26
 TextAlign: Right

Time to add the buttons. You need 10 buttons for the numbers 0 to 9. Add the first button to the form, and set the following properties for it:

Name: btnZero
Font: Microsoft Sans Serif, Bold, 12
Location: 143, 378
Size: 49, 40
Text: 0

This is the zero button, which goes at the bottom. Add a new button to your form and set the following properties for it:

Name: btnOne
Font: Microsoft Sans Serif, Bold, 12
Location: 66, 159
Size: 49, 40
Text: 1

And easier way to add new buttons is to copy and paste them. Click on **btnOne** to select it. Right click the button and select **Copy** from the menu that appears. Now click anywhere on the form. Right click again, and select **Paste**. A new button will appear with the number 1 on it. Have a look at the properties window, though, and you'll see that the new button has the Name **button1**. Change it to **btnTwo**. Then change the Text property to 2. Drag it in to position next to your number 1 button.

Add the other number buttons in the same: Copy, Paste, change the **Name** and the **Text** properties. For the other number buttons, use the following for the Name properties: btnThree, btnFour, btnFive, etc. Position your buttons like ours.

Add a new button for the Point symbol. Give it the Name **btnPoint**, and type a full stop (period) for the Text property. Change the Font property, if you think it's too small.

Only three buttons to go. So add three more buttons, and use the following properties:

Plus Button

| | |
|------------------|--------------------------------|
| Name: | btnPlus |
| Font: | Microsoft Sans Serif, Bold, 12 |
| Location: | 324, 159 |
| Size: | 49, 40 |
| Text: | + |

Equals Button

| | |
|------------------|--------------------------------|
| Name: | btnEquals |
| Font: | Microsoft Sans Serif, Bold, 12 |
| Location: | 324, 230 |
| Size: | 49, 40 |
| Text: | = |

Clear Button

| | |
|------------------|-------------------------------|
| Name: | btnClear |
| Font: | Microsoft Sans Serif, Bold, 8 |
| Location: | 324, 305 |
| Size: | 49, 40 |
| Text: | Clear |

Change the locations, though, if they don't match the alignment for your own buttons. But you've now completed the design of your calculator. Save your hard work, and we can begin the coding.

Code the Number Buttons

Before we get to the code, let's just go through how our calculator is going to work:

1. Click the number buttons. This will be the first number in the addition.
2. The first number you want to add will then appear in the text box
3. Click the Plus button to tell the calculator you want to add
4. The first number will disappear from the text box, ready for the second number
5. Click the number buttons again to add the second number
6. Click the equals button and the answer appears in the text box

The first task on the list is to get the number to appear in the text box when a number button is clicked. To do this, double click your number 1 button to get at the code.

The numbers on the buttons were put there by changing the text property. So the only thing we need to do is to access this text property. We can then use the button text as the text Property for the text box. Add the following line for your btnOne code:

```
txtDisplay.Text = btnOne.Text;
```

This says, “Make the Text in the text box the same as the Text that’s on the button”. Remember: whatever is on the right of the equals sign gets stored in whatever is on the left of the equals sign.

Run your programme and try it out. Click the number 1 button and it will appear in your text box. Click your number 1 a few times and what do you notice? You might think that clicking the number 1 button twice, for example, will cause the text box to display 11, and not 1. After all, you clicked it twice, so why shouldn’t two number 1’s appear?

The reason is doesn’t is because you haven’t told C# to keep the value that was already there. Each time you click the button, C# is starting afresh – it doesn’t know what was in there before, and discards the number that you previously stored.

Halt your programme and return to your code. Change your line to this:

```
txtDisplay.Text = txtDisplay.Text + btnOne.Text;
```

This line is easier to read if you just look at the part after the equals sign. Which is this:

```
txtDisplay.Text + btnOne.Text;
```

When you're working with text, the plus symbol doesn't mean add – it means concatenate (you learned about this in the previous section when working with strings).

So C# will join the text in the text box with the text on the button. After it has finished doing this, it will store the answer to whatever is on the left of the equals sign. In this case, it's not a variable but the text property of the text box.

Run your programme again. Click the number 1 button a few times. You should find that the number one will appear in the text box more than once.

Halt the programme and return not to your code but to the form itself. (If you can't see your form, right-click **Form1.cs** in the Solution Explorer on the right. From the menu that appears, select **View Designer**.)

Now double click button 2, and add the following code:

```
txtDisplay.Text = txtDisplay.Text + btnTwo.Text;
```

The only thing that's different is the name of the button – **btnTwo** instead of **btnOne**. The rest is the same.

Do the same for the rest of your buttons, changing the name of the button each time. (You can copy and paste your code to save time.)

But your coding window should look like this, when you've finished:

```
private void btnOne_Click(object sender, EventArgs e)
{
    txtDisplay.Text = txtDisplay.Text + btnOne.Text;
}

private void btnTwo_Click(object sender, EventArgs e)
{
    txtDisplay.Text = txtDisplay.Text + btnTwo.Text;
}

private void btnThree_Click(object sender, EventArgs e)
{
    txtDisplay.Text = txtDisplay.Text + btnThree.Text;
}

private void btnFour_Click(object sender, EventArgs e)
{
    txtDisplay.Text = txtDisplay.Text + btnFour.Text;
}

private void btnFive_Click(object sender, EventArgs e)
{
    txtDisplay.Text = txtDisplay.Text + btnFive.Text;
}

private void btnSix_Click(object sender, EventArgs e)
{
    txtDisplay.Text = txtDisplay.Text + btnSix.Text;
}

private void btnSeven_Click(object sender, EventArgs e)
{
    txtDisplay.Text = txtDisplay.Text + btnSeven.Text;
}

private void btnEight_Click(object sender, EventArgs e)
{
    txtDisplay.Text = txtDisplay.Text + btnEight.Text;
}

private void btnNine_Click(object sender, EventArgs e)
{
    txtDisplay.Text = txtDisplay.Text + btnNine.Text;
}

private void btnZero_Click(object sender, EventArgs e)
{
    txtDisplay.Text = txtDisplay.Text + btnZero.Text;
}
```

Run your programme again, and click all ten of your buttons. Make sure that each number appears in the text box when its button is clicked.

Return to your form and double click the Clear button. Add the following line:

txtDisplay.Clear();

After the full stop, you type the word **Clear**, followed by a pair of round brackets. **Clear** is a method you can use on text boxes. As its name suggests, it will clear the text box, leaving it blank.

Run your programme again, click a few numbers, then try your **Clear** button. The numbers should disappear from the text box.

The Plus Button

So we've got the numbers to appear in the text box when a button is clicked. The next thing we need to do is grab that number and store it somewhere. We'll then use this number when the equals button is clicked. (The equals button will do the addition sum, not the plus button.)

The plus button, then, needs to grab the number from the text box. Because it's text, we need to convert it to a number. We'll then store that number in a variable. The only other line of code we'll need for the Plus button is to clear the text box, ready for the second number.

The first number needs to be stored in a variable. We'll use the double type of variable. That way, we can have really big numbers with a “point something” at the end.

So that all the buttons in the programme can see this variable, it needs to be set outside of any button code. So we can't do this:

```
private void btnOne_Click(object sender, EventArgs e)
{
    double total1 = 0;
}
```

If you set up a variable inside of a button only this button will be able to do anything with the variable. This is known as **scope**. Because we've set up the variable inside of the button code, we've given it **local scope**: it can only be seen inside of the curly brackets. To make the variable accessible to all the buttons, we need to give it what's known as **global scope**. This is fairly easy – just set it up outside of any buttons. Like this:

```
double total1 = 0;

private void btnPlus_Click(object sender, EventArgs e)
{
}
```

Now the variable is outside of the curly brackets, making it available to all the buttons on our form. So set up that variable in your own code.

For the **btnPlus** code itself, add the following two lines:

```
double total1 = 0;

private void btnPlus_Click(object sender, EventArgs e)
{
    total1 = total1 + double.Parse(txtDisplay.Text);
    txtDisplay.Clear();
}
```

All we're doing here is getting the text from the text box, converting it to a **double** number, and then storing it in the **total1** variable. Notice that we've also done this:

total1 = total1 +

Just like we did for the number buttons, we need to keep whatever was in the **total1** variable. You need to do this in case you want to add more than two numbers. If you didn't keep what was in the **total1** variable, C# would "forget" what was in it, and start afresh. This technique is so common in programming that a shorthand way of doing this is usually implemented:

```
total1 += double.Parse(txtDisplay.Text);
```

So instead of repeating the variable name, you just use a plus symbol and equals symbol together (**+ =**). The above line does exactly the same thing as this:

```
total1 = total1 + double.Parse(txtDisplay.Text);
```

But whichever way you choose to retain a value in a variable, all you're saying is "Keep whatever is already in the variable, and add something else".

After storing the number from the text box, we need to clear it:

txtDisplay.Clear();

Once the text box is cleared, a second number can be selected by clicking the number buttons.

The Equals Button

The Equals button is where the action takes place. This is where we will do the actual addition.

To store the answer to the addition, we'll need another variable:

```

double total1 = 0;
double total2 = 0;

private void btnPlus_Click(object sender, EventArgs e)
{
    total1 = total1 + double.Parse(txtDisplay.Text);
    txtDisplay.Clear();
}

```

So set up a **total2** variable in your code, as we've done above.

Return to your Form, and double click the Equals button to get at your code. Now add the following three lines of code:

```

total2 = total1 + double.Parse( txtDisplay.Text );
txtDisplay.Text = total2.ToString();
total1 = 0;

```

The first line should look familiar:

```
total2 = total1 + double.Parse(txtDisplay.Text);
```

To the right of the equals sign, we're doing the same thing as before:

```
total1 + double.Parse(txtDisplay.Text);
```

The difference is before the equals sign: we're now storing it in the **total2** variable:

```
total2 = total1 + double.Parse(txtDisplay.Text);
```

In other words, get the number from the text box, convert it to a double variable, add it to whatever is in **total1**. When all this is worked out, store the answer in the variable called **total2**.

The second line of code was this:

```
txtDisplay.Text = total2.ToString();
```

On the right of the equals sign, we're converting the **total2** variable **To a String**. This is so that it can be displayed as Text in the text box.

The third line of code resets the **total1** variable to zero:

```
total1 = 0;
```

This is so a new sum can be calculated.

Time to try out your calculator. Use it to calculate the following:

10 + 25
36 + 36
10 + 10 + 10

Of course, you can do these sums in your head. But make sure that your calculator gets its sums right before going any further. Click your **Clear** button to start a new addition. When you're sure you understand what is going on with the code, try this exercise.

Exercise C

You have not yet written any code for the **btnPoint** button. This means that you can't have numbers like 10.5 or 36.7 in your additions. Write code to solve this. (Hint: you only need one line of code.)

[Answer to Exercise C](#)

In the next part, we're going to move on to something called Conditional Logic. We'll use Conditional Logic to complete the calculator. You will then be able to use it to divide, subtract, and multiply. So make sure you save the work you've done so far!

Conditional Logic

Conditional Logic is all about the IF word. In fact, it's practically impossible to programme effectively without using IF. You can write simple programmes like our calculator. But for anything more complicated, you need to get the hang of Conditional Logic.

As an example, take the calculator programme you have just written. It only has a Plus button. We'll be adding a second button soon, a Subtract button. Now, you can't say beforehand which of the two buttons your users will click. Do they want to add, or subtract? You need to be able to write code that does the following:

**IF the Plus button was clicked, add up
IF the Minus button was clicked, subtract**

You can rearrange the two statements above.

**Was the Plus button clicked? Yes, or No?
Was the Minus button clicked? Yes, or No?**

So the answer for each is either going to be Yes, or No – the button is either clicked, or not clicked.

IF Statements

To test for YES or NO values, you can use an IF statement. You set them up like this:

```
if()  
{  
}
```

So you start with the word “if” (in lowercase), and type a pair of round brackets. In between the round brackets, you type what you want to check for (Was the button clicked?). After the round brackets, it's convenient (but not strictly necessary) to add a pair of curly brackets. In between your curly brackets, you type your code. Your code is what you want to happen IF the answer to your question was YES, or IF the answer was NO. Here's a coding example:

```
bool buttonClicked = true;  
  
if(buttonClicked == true)  
{  
    MessageBox.Show("The button was clicked");  
}
```

Notice the first line of code:

```
bool buttonClicked = true;
```

This is a variable type you haven't met before – **bool**. The **bool** is short for Boolean. You use a Boolean variable type when you want to check for **true** or **false** values (YES, or NO, if you prefer). This type of variable can only ever be true or false. The name of the **bool** variable above is **buttonClicked**. We've set the value to true.

The next few lines are our IF Statement:

```
if (buttonClicked == true)
{
    MessageBox.Show("The button was clicked");
}
```

The double equals sign (`==`) is something else you need to get used to when using IF Statements. It means “Has a value of”. The double equals sign is known as a Conditional Operator. (There are a few others that you'll meet shortly.) But the whole of the line reads:

“**IF** `buttonClicked` has a value of **true**”

If you miss out one of the equals signs, you'd have this:

```
if (buttonClicked = true)
```

What you're doing here is assigning a value of true to the variable **buttonClicked**. It's not checking if **buttonClicked** “Has a value of” **true**. The difference is important, and will cause you lots of problems if you get it wrong!

In between the curly brackets of the IF statement, we have a simple `MessageBox` line. But this line will only get executed IF **buttonClicked** has a value of **true**.

Let's try it out. Start a new project for this (File > New project). Add a button to your new form, and set the Text property to “IF Statement”. Double click the button, and add the code from above. So your coding window will look like this:

```
private void button1_Click(object sender, EventArgs e)
{
    bool buttonClicked = true;

    if (buttonClicked == true)
    {
        MessageBox.Show("The button was clicked");
    }
}
```

Run your programme and click the button. You should see the message box. Now halt the programme and change this line:

```
bool buttonClicked = true;
```

to this

```
bool buttonClicked = false;
```

So the only change is from **true** to **false**. Run your programme again, and click the button. What happens? Nothing!

The reason that nothing happens is that our IF Statement is checking for a value of **true**:

```
if (buttonClicked == true)
```

C# will execute the code between the curly bracket IF, and only IF, **buttonClicked** has a value of **true**. Since you changed the value to **false**, it doesn't bother with the MessageBox in between the curly brackets, but moves on instead.

Else

You can also say what should happen if the answer was **false**. All you need to do is make use of the **else** word. You do it like this:

```
if (buttonClicked == true)
{
}
else
{
}
```

So you just type the word **else**, after the curly brackets of the IF Statement. And then add another pair of curly brackets. You then write your code for what should happen if the IF Statement was **false**. Change your code to this:

```
if (buttonClicked == true)
{
    MessageBox.Show("buttonClicked has a value of true");
}
else
{
    MessageBox.Show("buttonClicked has a value of false");
}
```

So the whole thing reads:

“IF it’s true that **buttonClicked** has a value of **true**, do one thing. If it’s not true, do another thing.”

Run your programme, and click the button. You should see the second MessageBox display. Halt the programme and change the first line back to **true**. So this:

```
bool buttonClicked = true;
```

instead of this:

```
bool buttonClicked = false;
```

Run the programme again, and click the button. This time, the first message box will display.

The whole point of using **IF ... Else** Statements, though, is to execute one piece of code instead of some other piece of code.

Else ... IF

Instead of using just the **else** word, you can use **else if**, instead. If we use our calculator as an example, we’d want to do this:

```
bool plusButtonClicked = true;
bool minusButtonClicked = false;

if (plusButtonClicked == true)
{
    //WRITE CODE TO ADD UP HERE
}
else if (minusButtonClicked == true)
{
    //WRITE CODE TO SUBTRACT HERE
}
```

So the code checks to see which button was clicked. If it’s the Plus Button, then the first IF Statement gets executed. If it’s the Minus Button, then the second IF Statement gets executed.

But **else if** is just the same as **if**, but with the word **else** at the start.

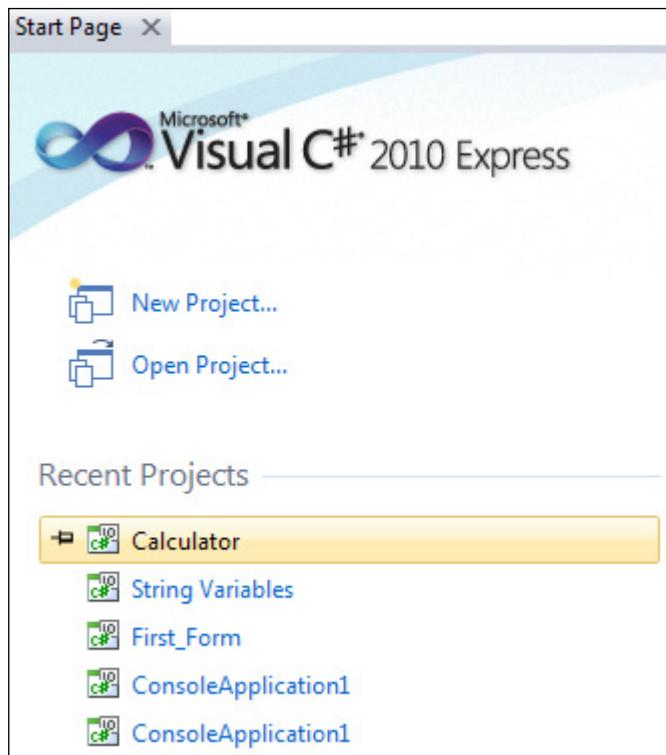
In fact, we can now add a minus button to our calculator. We’ll use **else if**.

So open up your calculator project again. To do this, click the link on the Start Page Tab in Visual Studio Express. If you can't see your Start Page tab, click its icon at the top of the C# software:



(In version 2012, click **View > Start Page** from the menu at the top.)

You should then see a section headed **Recent Projects**:



Look for your calculator project here. You can also click **File > Recent Projects** from the menu bar at the top.

If both of those fail, click **File > Open Project**. Navigate to where you saved your project. Open up the file that ends in **.sln**.

With your calculator project open, add a new button. Set the following properties for it in the Properties Window:

| | |
|------------------|--|
| Name: | btnMinus |
| Font: | Microsoft Sans Serif, 16, Bold |
| Location: | Move it to the right of your Plus button |
| Size: | 49, 40 |
| Text: | – |

Now double click your Minus button to get at its code. Add the following two Boolean variables outside of the Minus button code, just above it:

```
bool plusButtonClicked = true;
bool minusButtonClicked = false;
```

Your coding window will then look like this:

```
bool plusButtonClicked = false;
bool minusButtonClicked = false;

private void btnMinus_Click(object sender, EventArgs e)
{
```

Now add the following code inside of the Minus button:

```
total1 = total1 + double.Parse(txtDisplay.Text);
txtDisplay.Clear();

plusButtonClicked = false;
minusButtonClicked = true;
```

Your coding window will then look like the one below:

```
bool plusButtonClicked = false;
bool minusButtonClicked = false;

private void btnMinus_Click(object sender, EventArgs e)
{
    total1 = total1 + double.Parse(txtDisplay.Text);
    txtDisplay.Clear();

    plusButtonClicked = false;
    minusButtonClicked = true;

}
```

All we've done here is to set up two Boolean variables. We've set them both to **false** outside of the code. (They have been set up outside of the code because other buttons need to be able to use them; they have been set to false because no button has been clicked yet.) When the Minus button is clicked, we'll set the Boolean variable **minusButtonClicked** to **true** and the **plusButtonClicked** to **false**.

But the first two lines are exactly the same as for the Plus button:

```
total1 = total1 + double.Parse(txtDisplay.Text);
txtDisplay.Clear();
```

The first line just moves the numbers from the text box into the **total1** variable. The second line clears the text box.

Now access the code for your Plus button. Add two lines of code to the end:

```
private void btnPlus_Click(object sender, EventArgs e)
{
    total1 = total1 + double.Parse(txtDisplay.Text);
    txtDisplay.Clear();

    plusButtonClicked = true;
    minusButtonClicked = false;
}
```

So the only thing you are adding is this:

```
plusButtonClicked = true;
minusButtonClicked = false;
```

The Plus button resets the Boolean variables. This time, **plusButtonClicked** gets set to **true**, and **minusButtonClicked** gets set to **false**. It was the other way round for the Minus button.

The reason we're resetting these Booleans variables is because we can use them in an **if ... else if** statement. We can add up if the **plusButtonClicked** variable is true, and subtract if **minusButtonClicked** is true.

We'll still do the calculating in the Equals button. So change your equals button to this:

```
private void btnEquals_Click(object sender, EventArgs e)
{
    if (plusButtonClicked == true)
    {
        total2 = total1 + double.Parse(txtDisplay.Text);
    }
    else if (minusButtonClicked == true)
    {
        total2 = total1 - double.Parse(txtDisplay.Text);
    }

    txtDisplay.Text = total2.ToString();
    total1 = 0;
}
```

We're using Conditional Logic to decide which of the two buttons was clicked. The first if statement checks if the **plusButtonClicked** variable is true. If it is, then the addition gets done (this is exactly the same as before). If the first IF Statement is false, then C# moves down to the **else if** statement. If **minusButtonClicked** is true, then the subtraction gets done instead. The only difference between the addition and subtraction lines is the Operator symbols: a plus (+) instead of a minus (-).

The final two lines of code are the same as before – convert the number to text and display it in the text box, and then reset the total1 variable to zero. Run your calculator and try it out. You should be able to add and subtract.

Exercise D

Finish your calculator by adding Divide and Multiply buttons to your form. Write the code to make your calculator Divide and Multiply.

For this exercise, you're just adding two more Boolean variables to your code. You can then add more **else if** statements below the ones you already have. You'll also need to add two more lines to the code for your four Operator buttons. These two lines need to reset the Boolean variables to either true or false. For example, here's the code for the Minus button to get you started:

```
bool plusButtonClicked = false;
bool minusButtonClicked = false;
bool divideButtonClicked = false;
bool multiplyButtonClicked = false;

private void btnMinus_Click(object sender, EventArgs e)
{
    total1 = total1 + double.Parse(txtDisplay.Text);
    txtDisplay.Clear();

    minusButtonClicked = true;
    plusButtonClicked = false;
    divideButtonClicked = false;
    multiplyButtonClicked = false;
}
```

So we now have four Boolean variables outside the button code, one for the plus button, one for minus button, one for divide button, and one for the multiply button. When you click a button, its Boolean variable gets set to true.

Do the same for the other three buttons. Then write your **else if** statements. This is quite a tricky exercise, though. Probably your hardest so far!

[Answer to Exercise D](#)

Switch Statements

An easier way to code the calculator is by using a switch statement instead of an else if statement. A switch statement allows you to check which of more than one option is true. It's like a list of if statements. The structure of a switch statement looks like this:

```
switch ()
{
    case "Your_Test_1":
        //YOUR CODE HERE
        break;
    case "Your_Test_2":
        //more CODE HERE
        break;
}
```

After the word “switch”, you type a pair of round brackets. In between the round brackets, you type what you want to check for. You are usually testing what is inside of a variable. Then type a pair of curly brackets. In between the curly brackets, you have one **case** for each possible thing that your variable can contain. You then type the code that you want to execute if that particular case is true. After your code, type the word **break**. This enables C# to break out of the Switch Statement altogether.

We'll use our calculator as a coding example.

Our four buttons set a Boolean variable to either true or false. Instead of doing this, we could have the buttons put a symbol into a string variable. Like this:

```
string theOperator;

private void btnPlus_Click(object sender, EventArgs e)
{
    total1 = total1 + double.Parse(txtDisplay.Text);
    txtDisplay.Clear();

    theOperator = "+";
}
```

So the last line of code puts the + symbol into a string variable we've called **theOperator**. It will only do this if the button is clicked. The other buttons can do the same. We can then use a switch statement to check what is in the variable we've called **theOperator**. This will tell us which button was clicked:

```
switch (theOperator)
{
    case "+":
        //ADD UP HERE
        break;
    case "-":
        //SUBTRACT HERE
        break;
    case "*":
        //MULTIPLY HERE
        break;
}
```

```

case "/":
    //DIVIDE HERE
    break;
default :
    //DEFAULT CODE HERE
    break;
}

```

In between the round brackets after the word “switch”, we’ve typed the name of our variable (**theOperator**). We want to check what is inside of this variable. It will be one of four options: +, -, *, /. So after the first **case**, we type a plus symbol. It’s in between double quotes because it’s text. You end a **case** line with a colon:

```
case "+" :
```

The code to add up goes on a new line. After the code, the **break** word is used. So what you’re saying is:

“If it’s the **case** that **theOperator** holds a “+” symbol, then execute some code”

We have three more case parts to the switch statement, one for each of the math symbols. Notice the addition of this, though:

```

default :
    //DEFAULT CODE HERE
    break;

```

You use **default** instead of **case** just “in case” none of the options you’ve thought of are what is inside of your variable. You do this so that your programme won’t crash!

Exercise E

Adapt your calculator code so that it uses a switch statement instead of **if ... else if** Statements. You’ll need to get rid of all your Boolean variables!

[Answer to Exercise E](#)

Switch statements can be quite tricky to get the hang of. But they are very useful if you want to test a variable for a list of possible things that could be in the variable.

Conditional Operators

You've already met one Conditional Operator, the double equals sign (`==`). You use this in IF Statements when you want to check if a variable "has a value of" something:

```
if( myVariable == 10)
{
    //EXECUTE SOME CODE HERE
}
```

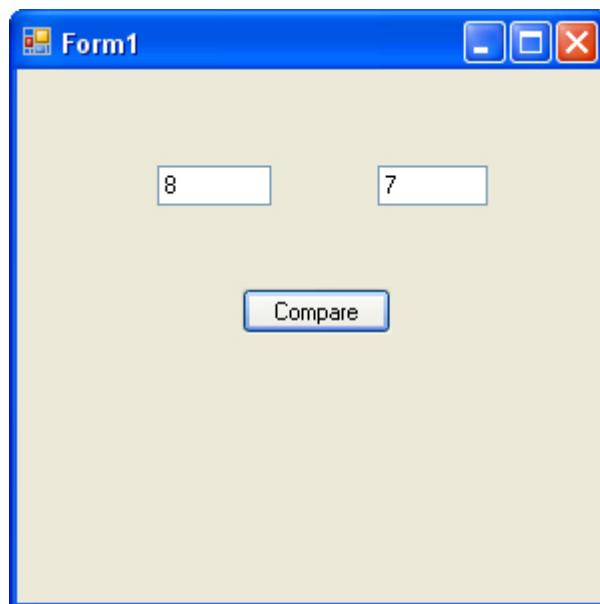
So the above line reads, "IF whatever is inside of **myVariable** has a value of 10, execute some code."

Other Conditional Operators you'll use when you're coding are these:

| | |
|----|---------------------------------|
| > | Greater Than |
| < | Less Than |
| >= | Greater Than or Equal To |
| <= | Less Than or Equal To |
| != | Not Equal To |
| ! | Not |
| && | And |
| | Or |

Because you need to learn these Operators, let's get some practice with them.

Start a new project. Add two text boxes and a button to your form. Resize the text boxes and type 8 as the Text property for the first text box, and 7 as the Text property for the second text box. Set the Text property for the button to the word "Compare". Your form will then look like this:



Double click the button to get at the coding window. What we'll do is to get the numbers from the text boxes and test and compare them. So the first thing to do is to set up some variables:

```
int firstNumber;
int secondNumber;
```

Then get the text from the text boxes and store them in the variables (after converting them to integers first.)

```
firstNumber = int.Parse(textBox1.Text);
secondNumber = int.Parse(textBox2.Text);
```

What we want to do now is to compare the two numbers. Is the first number bigger than the second number? To answer this, we can use an IF Statement, along with one of our new Conditional Operators. So add this to your code:

```
if (firstNumber > secondNumber)
{
    MessageBox.Show("The first number was greater than the
                    second number");
}
```

Your coding window will then look like this (our message box above is only on two lines because it can't all fit on this page):

```
private void button1_Click(object sender, EventArgs e)
{
    int firstNumber;
    int secondNumber;

    firstNumber = int.Parse(textBox1.Text);
    secondNumber = int.Parse(textBox2.Text);

    if (firstNumber > secondNumber)
    {
        MessageBox.Show("The first number was greater than the second number");
    }
}
```

So in between the round brackets after **if**, we have our two variables. We're then comparing the two and checking to see if one is **Greater Than** (**>**) the other. If **firstNumber** is Greater Than **secondNumber** then the message box will display.

Run your programme and click your button. You should see the message box display. Type a 6 in the first text box, and click the button again. The message box won't display. It won't display because 6 is not greater than 7. The message box code is inside of the curly brackets of the IF Statement. And the IF Statement only gets executed if **firstNumber** is Greater Than **secondNumber**. If it's not, C# will just move on to the next line. You haven't got any more lines, so C# is finished.

Stop your programme and go back to your code. Add a new if statement below your first one:

```
if (firstNumber < secondNumber)
{
    MessageBox.Show("The first number was less than the
                    second number");
}
```

Again, our message box above is spread over two lines because there's not enough room for it on this page. Your message box should go on one line. But the code is just about the same! The thing we've changed is to use the Less Than symbol (`<`) instead of the Greater Than symbol (`>`). We've also changed the text that the message box displays.

Run your programme, and type a 6 in the first text box. You should see your new message box display. Now type an 8 in the first text box, and click your button. The first message box will display. Can you see why? If your programme doesn't work at all, make sure it is like ours in the image below:

```
int firstNumber;
int secondNumber;

firstNumber = int.Parse(textBox1.Text);
secondNumber = int.Parse(textBox2.Text);

if (firstNumber > secondNumber)
{
    MessageBox.Show("The first number was greater than the second number");
}

if (firstNumber < secondNumber)
{
    MessageBox.Show("The first number was less than the second number");
}
```

With your programme still running, type a 7 in the first box. You will then have a 7 in both text boxes. Before you click your button, can you guess what will happen?

The reason that nothing happens at all is because you haven't written any code to say what should happen if both numbers are equal. For that, try these new symbols:

`>=` Greater Than or Equal to

And these ones

`<=` Less Than or Equal to

Try these new Conditional Operators in place of the ones you already have. Change the text for your message boxes to suit. Run your code again. When you

click the button, both message boxes will display, one after the other. Can you see why this happens?

Another Conditional Operator to try is “Not Equal To” (!=). This is an exclamation mark followed by an equals sign. It is used like this:

```
if (firstNumber != secondNumber )
{
    //SOME CODE HERE
}
```

So, “IF firstNumber is not equal to secondNumber execute some code.”

You can even use the exclamation mark by itself. You do this when you want to test for a false value between the round brackets after **if**. It’s mostly used with Boolean values. Here’s an example:

```
bool testValue = false;

if (!testValue)
{
    MessageBox.Show("Value was false");
}
```

So the exclamation mark goes before the Boolean value you want to test. It is a shorthand way of saying “If the Boolean value is false”. You can write the line like this instead:

```
if (testValue == false)
```

But experienced programmers just use the exclamation mark instead. It’s called the NOT Operator. Or the “IF NOT true” Operator.

The final two Operators we’ll have a look at are these:

| | |
|-------------------|---------------------|
| && | (means AND) |
| | (means OR) |

These two are known as Logical Operators, rather than Conditional Operators (so is the NOT operator).

The two ampersand together (&&) mean AND. You use them like this:

```
bool isTrue = false;
bool isFalse = false;

if ( isTrue == false && isFalse == false )
{
}
```

You use the AND operator when you want to check more than one value at once. So in the line above, you're checking if both values are false. If and ONLY if both of your conditions are met will the code between curly brackets get executed. In the code above, we're saying this:

“If **isTrue** has a value of false AND if **isFalse** has a value of false then and only then executed the code between curly brackets.”

If **isTrue** is indeed true, for example, then any code between curly brackets won't get executed – they both have to be false, in our code.

You can test for only one condition of two being met. In which, use the OR (||) operator. The OR operators is two straight lines. These can be found above the back slash character on a UK keyboard, which is just to the left of the letter “Z”. (The | character is known as the pipe character.) You use them like this:

```
bool isTrue = false;
bool isFalse = false;

if ( isTrue == false || isFalse == false )
{
}
```

We're now saying this:

“If **isTrue** has a value of false OR if **isFalse** has a value of false then and only then executed the code between curly brackets.”

If just one of our variables is false, then the code in between curly brackets will get executed.

Try not to worry if you don't have a thorough grasp of all the Conditional Operators yet – you'll get the hang of them as you go along. But try the next exercise.

Exercise F

Write a small programme with a text box and a button. Add a label to ask people to enter their age. Use Conditional Logic to test how old they are. Display the following messages, depending on how old they are:

| | |
|----------------------|---------------------------------------|
| Less than 16 | “You’re still a youngster.” |
| Over 16 but under 25 | “Fame beckons!” |
| Over 25 but under 40 | “There’s still time.” |
| Over 40 | “Oh dear, you’ve probably missed it!” |

Only one message box should display, when you click the button. Here's some code to get you started:

```
int age;  
  
age = int.Parse(textBox1.Text);  
  
if (age < 17)  
{  
    MessageBox.Show("Still a youngster.");  
}
```

For the others, just add **else if** Statements, and more Condition Operators.

[Answer to Exercise F](#)

In the next section, we'll have a look at loops, which are another crucial hurdle to overcome in programming. By the end of the section, you'll have written your own times table programme.

Looping the Loop

Loops are an important part of any programming language, and C# is no different. A loop is a way to execute a piece of code repeatedly. The idea is that you go round and round until an end condition is met. Only then is the loop broken. As an example, suppose you want to add up the numbers one to ten. You could do it like this:

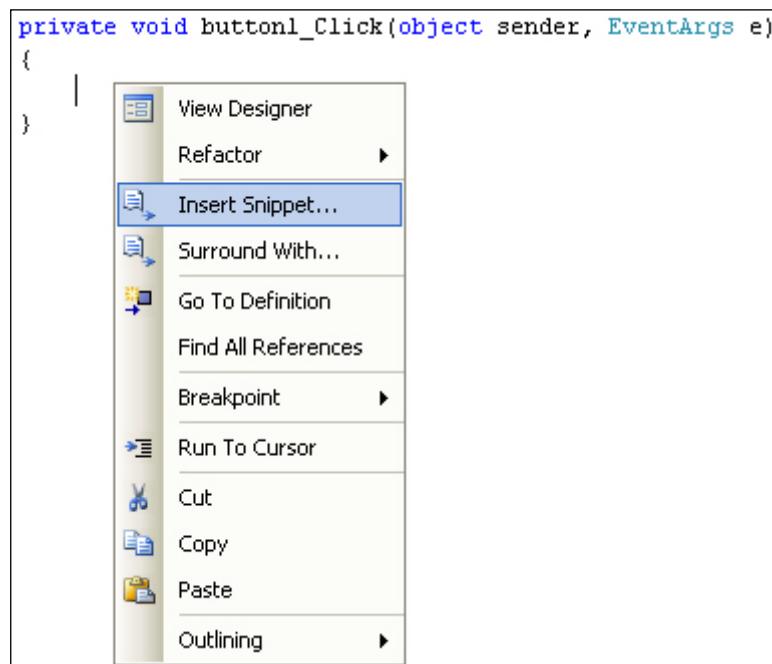
```
int answer;  
answer = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10;
```

And this would be OK if you only had 10 numbers. But suppose you had a thousand number, or ten thousand. You're certainly not going to want to type them all out! Instead, you use a loop to repeatedly add the numbers.

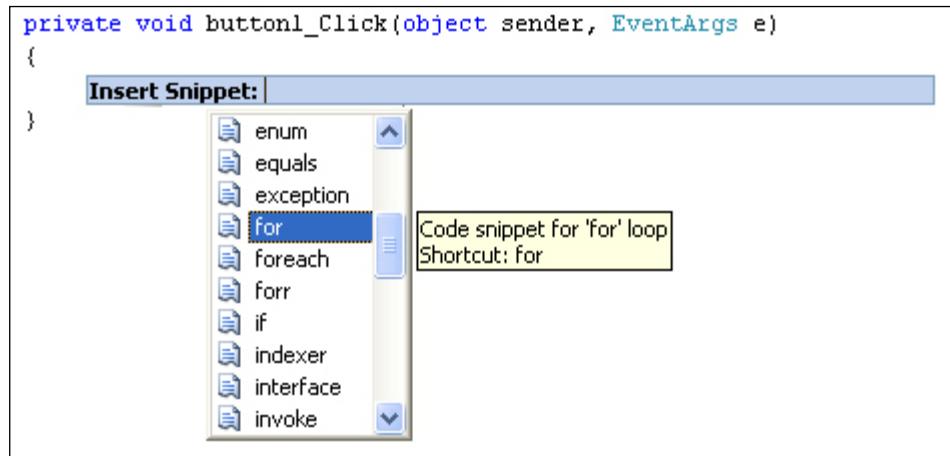
For Loops

The first type of loop we'll explore is called a **for** loop. Other types are **do** loops and **while** loops, which you'll meet shortly. But the **for** loop is the most common type of loop you'll need. Let's use one to add up the numbers 1 to 100.

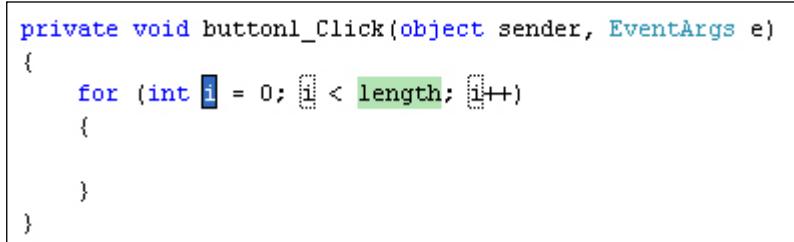
Start a new project by clicking **File > New Project** from the menu bars at the top of Visual Studio. Now add a button to the new form. Double click the button to get at the code. To quickly add a code stub for a loop, right click anywhere between the curly brackets of the button code. From the menu that appears, click on **Insert Snippet**:



When you click on Insert Snippet, you'll see a list of items (click the C# entry in version 2012, first):



Scroll down and double click on **for**. Some code is added for you:



It all looks a bit complicated, so we'll go through it. Here's the for loop without anything between the round brackets:

```
for ()
{
```

So you start with the word **for**, followed by a pair of round brackets. What you are doing between the round brackets is telling C# how many times you want to go round the loop. After the round brackets, you type a pair of curly brackets. The code that you want to execute repeatedly goes between the curly brackets.

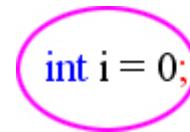
The default round-bracket code that C# inserts for you is this:

```
int i = 0; i < length; i++
```

There's three parts to the round-bracket code:

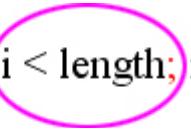
1. Which number do you want your loop to start at?
2. How many times do you want to go round and round?
3. How do you want to update each time round the loop?

Note that each of the three parts is separated by a semi-colon. Here's the first part:



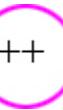
```
int i = 0; i < length; i++
```

And here's the second part:



```
int i = 0; i < length; i++
```

And here's the third part:



```
int i = 0; i < length; i++
```

Number 1 on the list above (Which number do you want to start at?) is this:

```
int i = 0;
```

What the default code is doing is setting up an **integer** variable called **i** (a popular name for loop variables.) It is then assigning a value of 0 to the **i** variable. It will use the value in **i** as the starting value of the loop. You can set up your starting variable outside the code, if you prefer. Like this:

```
int i

for (i = 0; i < length; i++)
{
```

```
}
```

So the variable called **i** is now set up outside the loop. We then just need to assign a value to the variable for the first part of the loop.

Number 2 on the list above (How many times do you want to go round and round?) was this:

```
i < length;
```

This, if you remember your Conditional Logic from the previous section, says “**i** is less than length”. But length is not a keyword. So you need to either set up a variable called length, or replace the word length with a number. So either this:

```
for (int i = 0; i < 101; i++)
```

Or this:

```
int length = 101;

for (int i = 0; i < length; i++)
{
}
```

In the first example, we've just typed the **i < 101**. In the second example, we've set up a variable called **length**, and stored 101 in it. We're then just comparing one variable to another, and checking that **i** is less than **length**:

i < length;

If **i** is less than **length**, then the end condition has NOT been met and C# will keep looping. In other words, "Keep going round and round while **i** is less than **length**."

But you don't need to call the variable **length**. It's just a variable name, so you can come up with your own. For example:

```
int endNumber = 101;

for (int i = 0; i < endNumber; i++)
{
}
```

Here, we've called the variable **endNumber** instead of **length**. The second part now says "Keep looping while **i** is less than **endNumber**".

Number 3 on the list above (How do you want to update each time round the loop?) was this:

i++

This final part of a **for** loop is called the Update Expression. For the first two parts, you set a start value, and an end value for the loop. But C# doesn't know how to get from one number to the other. You have to tell it how to get there. By typing **i++**, you are adding 1 to the value inside of **i** each time round the loop (called incrementing the variable). This:

variable_name++

is a shorthand way of saying this:

variable_name = variable_name + 1

All you are doing is adding 1 to whatever is already inside of the variable name. Since you're in a loop, C# will keep adding 1 to the value of **i** each time round the loop. It only stops adding 1 to **i** when the end condition has been reached (**i** is no longer less than **length**).

So to recap, you need a start value for the loop, how many times you want to go round and round, and how to get from one number to the other.

So your three parts are these:

for (Start_Value; End_Value; Update_Expression)

OK, time to put the theory into practice. Type the following for your button code:

```
int answer = 0;

for (int i = 1; i < 101; i++)
{
    answer = answer + i;
}

MessageBox.Show(answer.ToString());
```

Your coding window should then look like this, when you're finished:

```
private void button1_Click(object sender, EventArgs e)
{
    int answer = 0;

    for (int i = 1; i < 101; i++)
    {
        answer = answer + i;
    }

    MessageBox.Show(answer.ToString());
}
```

The actual code for the loop, the code that goes inside of the curly brackets, is this:

answer = answer + i;

This is probably the trickiest part of loops – knowing what to put for your code! Just remember what you're trying to do: force C# to execute a piece of code a set number of times. We want to add up the numbers 1 to 100, and are using a variable called **answer** to store the answer to the addition. Because the value in **i** is increasing by one each time round the loop, we can use this value in the addition. Here are the values the first time round the loop:

answer = answer + i;

0 = 0 + 1

The second time round the loop, the figures are these:

answer = answer + i;

1 = 1 + 2

The third time round the loop:

answer = answer + i;

3 = 3 + 3

And the fourth:

answer = answer + i;

6 = 6 + 4

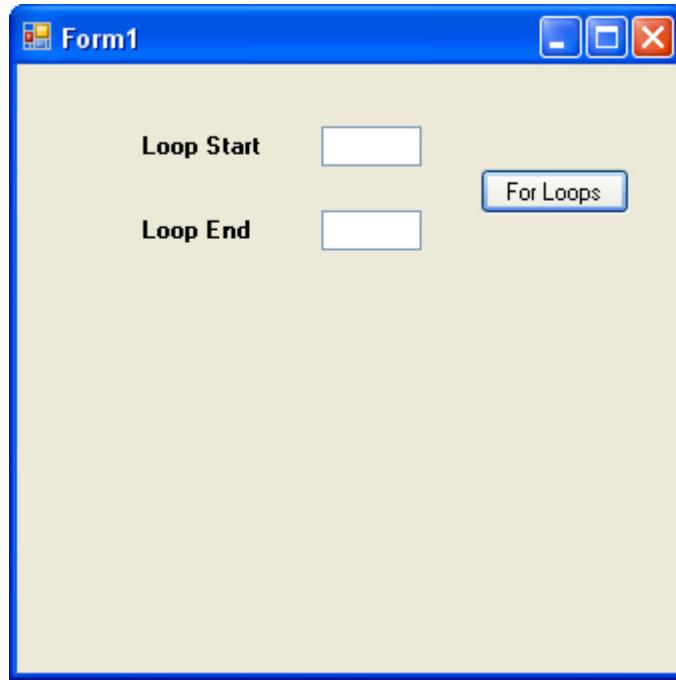
Notice how the value of **i** increases by one each time round the loop. If you first do the addition after the equals sign, the above will make more sense! (As an exercise, what is the value of **answer** the fifth time round the loop?)

Run your programme, and click the button. The message box should display an answer of 5050.

Loop Start Values and Loop End Values

In the code above, we typed the start value and end value for the loop. You can also get these from text boxes.

Add two text boxes to your form. Add a couple of labels, as well. For the first label, type **Loop Start**. For the second label, type **Loop End**. Your form will then look something like this:



What we'll do is to get the start value and end value from the text boxes. We'll then use these in our **for** loop.

So double click your button to get at the code. Set up two variables to hold the numbers from the text boxes:

```
int loopStart;  
int loopEnd;
```

Now store the numbers from the text boxes into the two new variables:

```
loopStart = int.Parse(textBox1.Text);  
loopEnd = int.Parse(textBox2.Text);
```

Now that we have the numbers from the text boxes, we can use them in the for loop. Change your for loop to this:

```
for (int i = loopStart; i < loopEnd; i++)  
{  
    answer = answer + i;  
}
```

The only thing you're changing here is the part between the round brackets. The first part has now changed from this:

```
int i = 1
```

to this:

```
int i = loopStart
```

So instead of storing a value of 1 in the variable called **i**, we've stored whatever is in the variable called **loopStart**. Whatever you type in the first text box is now used as the starting value of the loop.

For the second part, we've changed this:

i < 101

to this:

i < loopEnd

We're using the value stored inside of **loopEnd**. We're telling C# to keep looping if the value inside of the **i** variable is less than **loopEnd**. (Remember, because our Update Expression is **i++**, C# will keep adding 1 to the value of **i** each time round the loop. When **i** is no longer less than **loopEnd**, C# will stop looping.)

Run your programme and type 1 in the first text box and 10 in the second text box. Click your button. You should find that the message box displays an answer of 45.

Can you see a problem here? If you wanted to add up the numbers from 1 to 10, then the answer is wrong! It should be 55, and not 45. Can you see why 45 is displayed in the message box, and not 55? If you can't, stop a moment and try to figure it out.

(There is, of course, another problem. If you don't type anything at all in the text boxes, your programme will crash! It does this because C# can't convert the number from the text box and store it in the variable. After all, you can't expect it to convert something that's not there! You'll see how to solve this at the end of the chapter.)

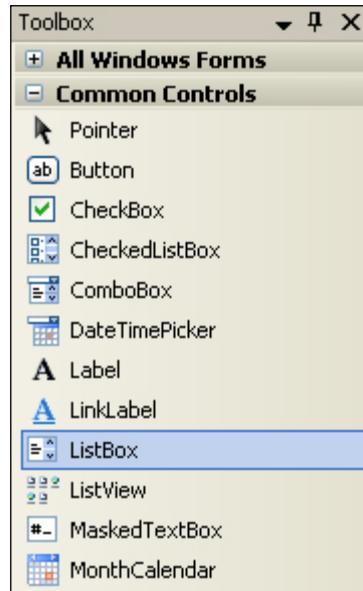
Times Table programme

We can now write a little times table programme. We'll use the text boxes to ask users to input a start number and end number. We'll use these to display the 10 times table. So if the user types a 1 into the first text box and a 5 into the second text box, we'll display this:

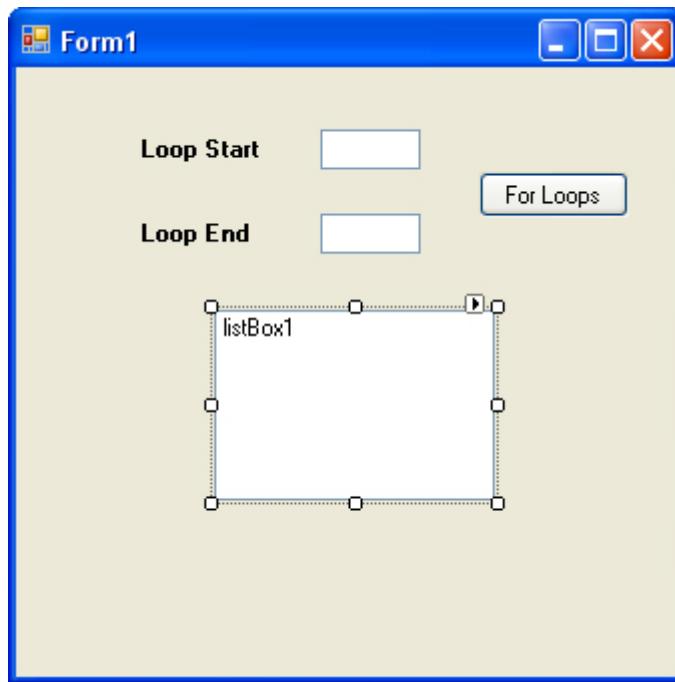
**1 times 10 = 10
2 times 10 = 20
3 times 10 = 30
4 times 10 = 40
5 times 10 = 50**

Instead of using a message box to display the results, we'll use a List Box. A list box, you will not be surprised to hear, is used to display lists of items. But it's

easier to show you what they do rather than explain. So use the Toolbox on the left of Visual Studio to add a list box to your form:



Resize your list box, and your form will now look something like ours below:

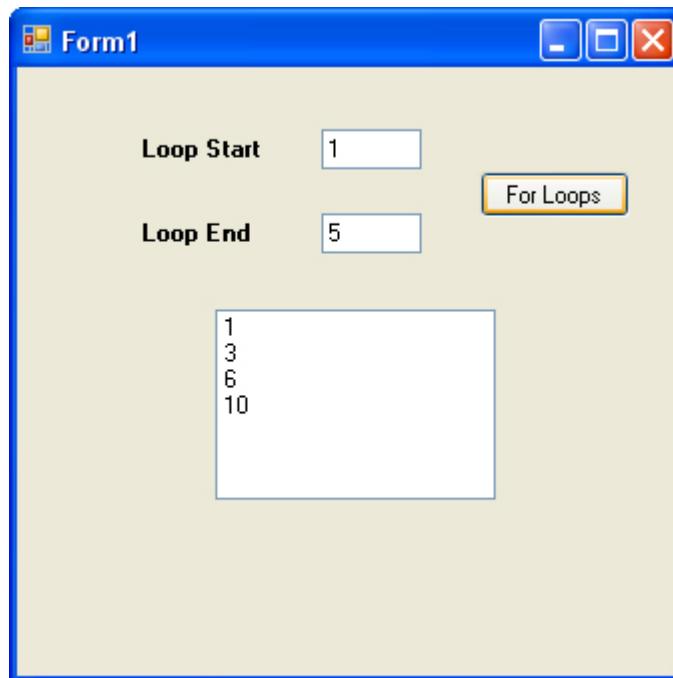


Double click your button to get at your code. Now add this line to your loop (the line to add is in bold):

```
for (int i = loopStart; i <= loopEnd; i++)
{
    answer = answer + i;
    listBox1.Items.Add( answer.ToString() );
}
```

So you start by typing the name of your list box. After a dot, you should see the IntelliSense list appear. Select **Items** from the list. **Items** is another property of list boxes. It refers to the items in your list. After the word **Items**, you type another dot. From the IntelliSense list, select the **Add** method. As its name suggests, the **Add** method adds items to your list box. Between the round brackets, you type what you want to add to the list of items. In our case, this was just the **answer**, converted to a **string**.

You can delete the message box line, if you like, because you don't need it. But run your programme and enter 1 in the first text box and 5 in the second text box. Click your button and your form should look like this:



The programme is supposed to add up the number 1 to 5, or whatever numbers were typed in the text boxes. The list box is displaying one **answer** for every time round the loop. However, it's only displaying 4 items. If you solved the problem as to why 45 was displayed in the message box, and not 55 then you'll already know why there are only four items in the list box. If you didn't, examine the first line of the for loop:

```
for (int i = loopStart; i < loopEnd; i++)
```

The problem is the second part of the loop code:

i < loopEnd

We're telling C# to go round and round while the value in **i** is **less than** the value in **loopEnd**. C# will stop looping when the values are equal. The value in **loopEnd** is 5 in our little programme. So we're saying this to C#, "Keep looping while the value in **i** is less than 5. Stop looping if it's 5 or more."

Cleary, we've used the wrong Conditional Operator. Instead of using the **less than** operator, we need ... well, which one do we need? Replace the < symbol with the correct one.

Exercise G

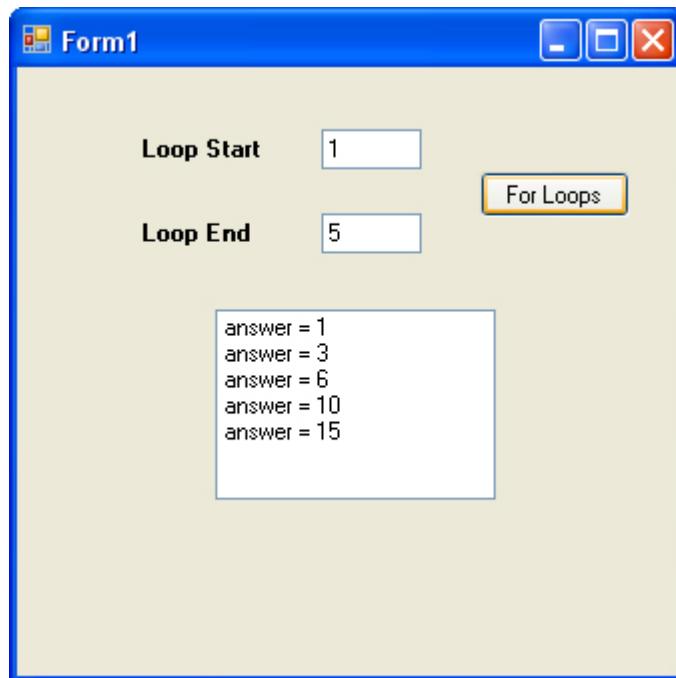
For some extra points, can you think of another way to solve the problem? One where you can keep the less than symbol?

[Answer to Exercise G](#)

To add some more information in your list box, change your line of code to this:

```
listBox1.Items.Add( "answer = " + answer.ToString( ) );
```

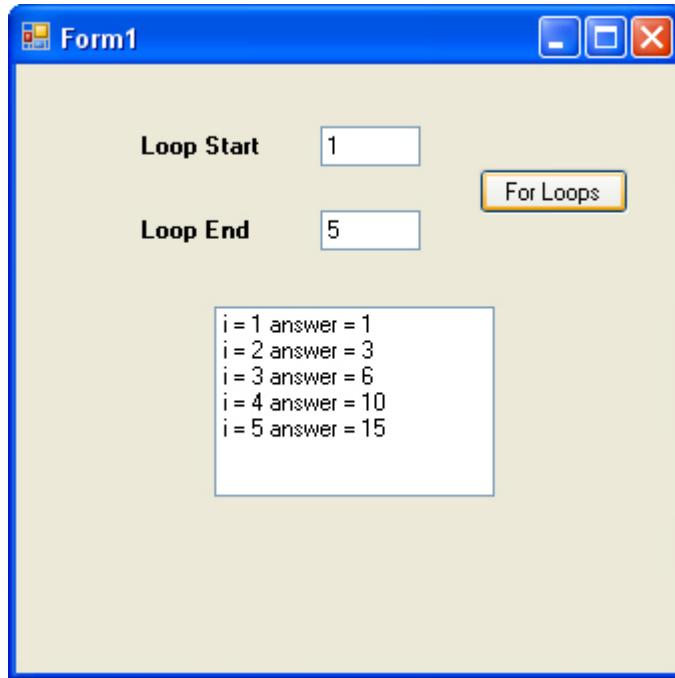
The thing to add is the text in bold above. We've typed some direct text "answer=" and followed this with the concatenation symbol (+). C# will then join the two together, and display the result in your list box. Run your programme again, and the list box will be this:



To make it even clearer, add some more text to your list box. Try this:

```
listBox1.Items.Add( "i = " + i + " answer = " + answer.ToString( ) );
```

Again, the text to add is in bold. Run your programme, and your list box will look like this:



We've now added the value of the variable called **i**. This makes it clear how the value of **i** changes each time round.

We now have all the ingredients to write the Times Table programme.

So return to your coding window. What we're going to do, remember, is to use a for loop to calculate and display the 10 times table. We need another variable, though, to hold the 10. So add this to your variables:

```
int multiplyBy = 10;
```

The only other thing we need to do is to change the code between the curly brackets of the **for** loop. At the moment, we have this:

```
answer = answer + i;
listBox1.Items.Add("i = " + i + " answer = " + answer.ToString());
```

Delete these two lines and replace them with these two:

```
answer = multiplyBy * i;

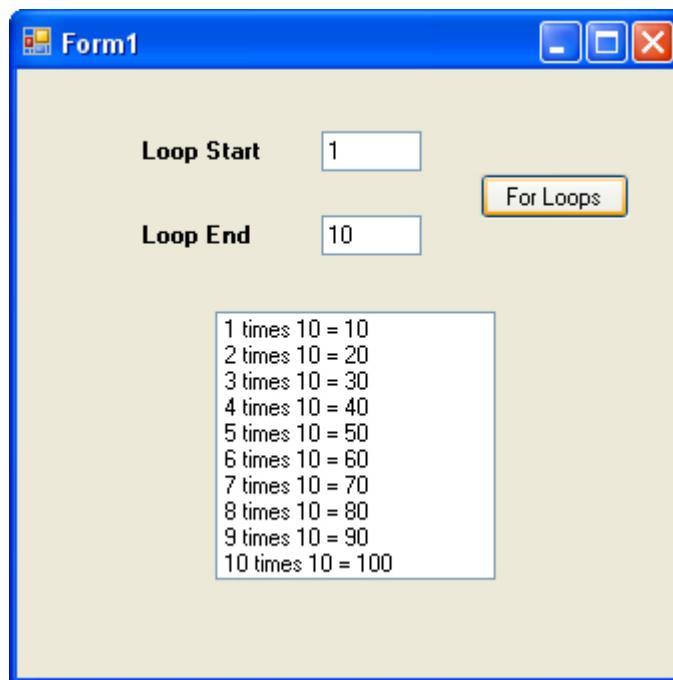
listBox1.Items.Add(i + " times " + multiplyBy + " = " +
    answer.ToString());
```

The list box line is a bit messy, and is spread over two lines in this book. Keep yours on one line. But the part between the round brackets is this:

i + " times " + multiplyBy + " = " + answer.ToString()

It's just a combination of variable names and direct text. The first line of the code, though, is a simple multiplication. We multiply whatever is inside of the variable called **multiplyBy** (which is 10) by whatever is inside of the variable called **i**. C# is adding 1 to the value of **i** each time round the loop, so the **answer** gets updated and then displayed in the list box.

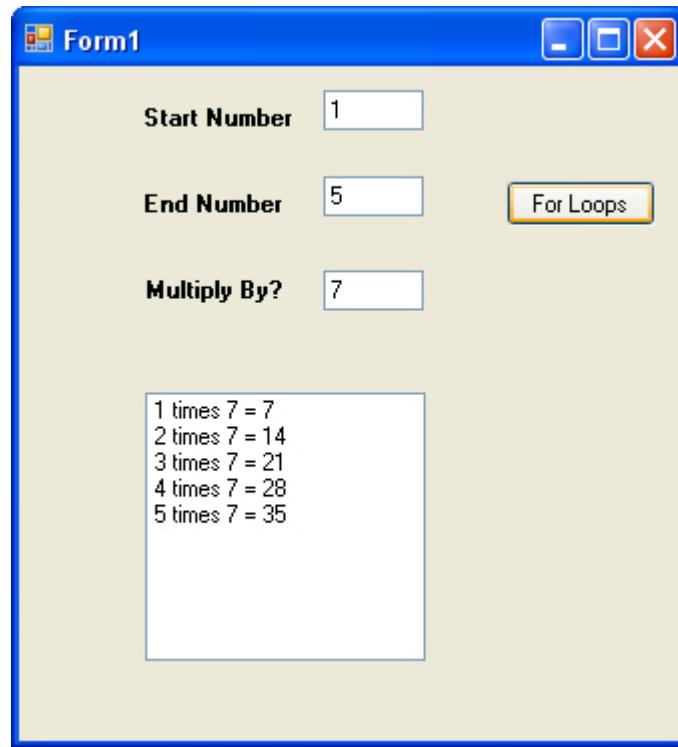
Run your programme. Enter 1 in the first text box and 10 in the second text box. Click you button to see the ten times table:



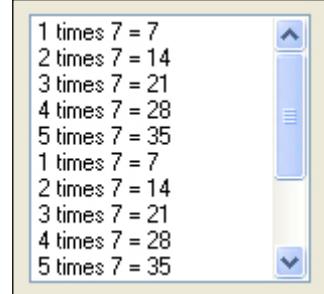
So with just a few lines of code, and the help of a **for** loop, we've created a programme that does a lot of work. Think how much more difficult this type of programme would be without looping.

Exercise H

At the moment, we're multiplying by 10 and, therefore, displaying the 10 times table. Add another text box to your form. Add a label that asks users which times table they want. Use this value in your code to display the times table for them. So if your user types a 7 into your new text box, clicking the button would display the 7 times table in the list box. Here's what your programme should look like when you're finished (we've change the first two labels, as well):



When you complete this exercise, click your button a few times. You'll notice (as long as you have numbers in the text boxes) that the list box doesn't clear itself. You'll have this in your list box:



So the new information is simply added to the end. To solve this, add the following line anywhere before your **for** loop, but inside of your button code.

```
listBox1.Items.Clear();
```

So instead of using the **Add()** method, you use the **Clear()** method. This will clear out all the items in a list box. But can you see why the line of code would be no good inside of the loop? Or after the loop?

[Answer to Exercise H](#)

But that's enough of **for** loop. We'll now briefly explore two other types of loops: **do** loops and **while** loops.

Do loops and While Loops

As well as using a **for** loop to repeatedly execute some code, you can use a Do loop or a While loop. We'll start with the Do Loop.

Do Loops

Whichever loop you use, the idea is the same: go round and round and execute the same code until an end condition is met. The difference with the Do and While loops is in the structure. Here's what the Do loop looks like:

```
do
{
    } while (true);
```

Notice where the semi-colon is, in the code above. It comes right at the end, after the round brackets. But you start with the word **do**, followed by a pair of curly brackets. After the curly brackets, you type the word **while**. After **while**, and in between some round brackets, you type your end condition. C# will loop round and round until the end condition between the round brackets is met. Only then will it bail out. Here's an example, using our times table programme (It assumes that the variable **i** has been set to a starting value for the loop):

```
do
{
    answer = multiplyBy * i;
    listBox1.Items.Add(answer.ToString());
    i++;
} while (i <= loopEnd);
```

So this time, we've used a Do Loop instead of a For Loop. The loop will go round and round **while** the value in the variable called **i** is less than or equal to the value in the variable called **loopEnd**. The other thing to notice here is we have to increment (add one to) the value in **i** ourselves (**i++**). We do this each time round the loop. If we didn't increment the value in **i** then it would always be less than **loopEnd**. We'd have then created an infinite loop, and the programme would crash. But we're really saying this:

“Keep **Doing** the code in curly brackets **while** **i** is less than or equal to **loopEnd**.”

While Loops

While loops are very similar in structure to Do loops. Here's what they look like:

```
while (true)
{
}
```

And here's the times table code again:

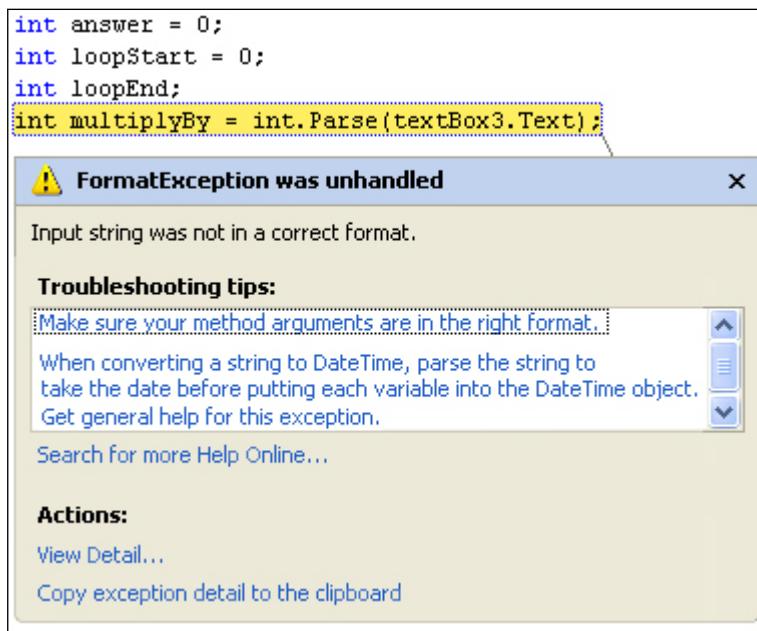
```
while (i <= loopEnd)
{
    answer = multiplyBy * i;
    listBox1.Items.Add(answer.ToString());
    i++;
}
```

While loops are easier to use than Do loops. If you look at the code above, you can see that the **while** part is now at the start, instead of at the end, like a Do Loop. The code you need to execute repeatedly still goes between curly brackets. And you still need a way for the loop to end (**i++**). The difference between the two is that the code in a Do loop will get executed at least once, because the while part is at the end. With the while part at the beginning, your end condition in round brackets can already be true (**i** might be more than **loopEnd**). In which case, C# will bail out immediately, and the code in curly brackets won't get executed at all.

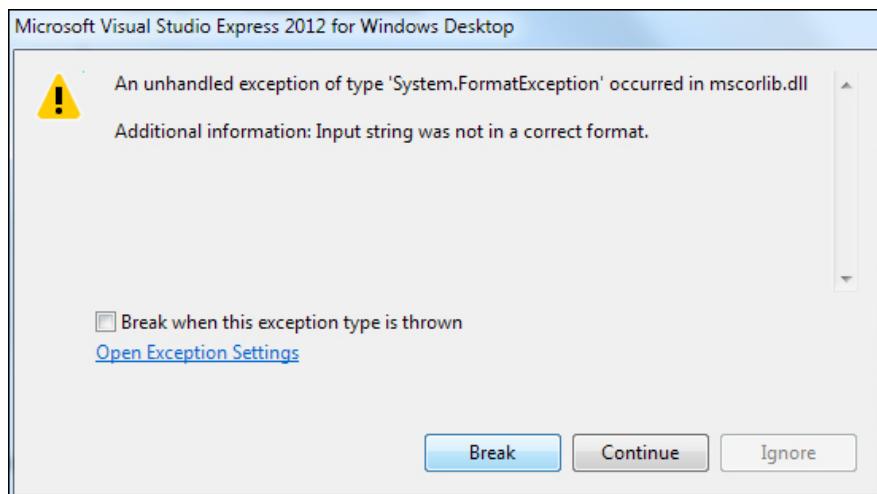
Deciding which loop to use can be quite tricky. Don't worry if you haven't fully understood how to use loops. You'll get lots more practice as you work your way through this book. But loops are difficult to get the hang of, and you shouldn't consider yourself a failure if you haven't yet mastered them. For now, we'll leave this complex subject, and end the section with a problem, and a solution.

Try Parse

There's a problem with the text boxes on your times table programme. If you don't type anything at all in the text boxes, your programme will crash! Try it out. Start your programme and leave the text boxes blank. Now click your button. You should get a strange and unhelpful error message:



C# 2012 will just give you this rather plain error:



C# is highlighting the offending line in yellow. It does this because it can't convert the numbers from the text box and store them in the variables. After all, you can't expect it to convert something that's not there! To remedy this, you can use a method called **TryParse**.

To convert the numbers from the text boxes to integers, you've been doing this:

```
loopStart = int.Parse(textBox1.Text);
```

So you've **Parsed** the number in the text box, and turned it into an **int**. But this won't check for blank text boxes, and it won't check to see if somebody typed, say, the word three instead of the number 3. What you need to do is to **Try** and Parse the data in the text box. So you ask C# if it can be converted into a number. If it can't, you display an error message for your users. Here's some code that

tries to parse the data from the first text box. It's a bit complex, so we'll go through it.

```

int outputValue = 0;
bool isNumber = false;

isNumber = int.TryParse(textBox1.Text, out outputValue);

if (!isNumber)
{
    MessageBox.Show("Type numbers in the text boxes");
}
else
{
    //REST OF CODE HERE
}

```

The first two lines set up some variables, an integer and a Boolean. The **outputValue** is needed for **TryParse**. You are trying to **output** a Boolean value (true or false) AND the string of text:

```
isNumber = int.TryParse(textBox1.Text, out outputValue);
```

So **isNumber** will be either true or false, depending on whether or not C# can convert the text box data into an integer. If you were trying to parse a **double** variable your code would be this, instead:

```

double outputValue = 0;
bool isNumber = false;

isNumber = double.TryParse(textBox1.Text, out outputValue);

```

The output value that you need is now a **double** (You're checking to see if C# can convert to a double value). The value of **isNumber** will still be either true or false (can it be converted or not?).

After using **TryParse**, you then need to check that true or false value:

```

if (!isNumber)
{
    MessageBox.Show("Type numbers in the text boxes");
}
else
{
    //REST OF CODE HERE
}

```

If you remember the lesson on Conditional Operators, you'll know that this line:

```
if (!isNumber)
```

Reads this:

If NOT true

If you prefer, you can write the line like this:

```
if (isNumber == false)
```

The line now reads:

“If **isNumber** has a value of **false**”

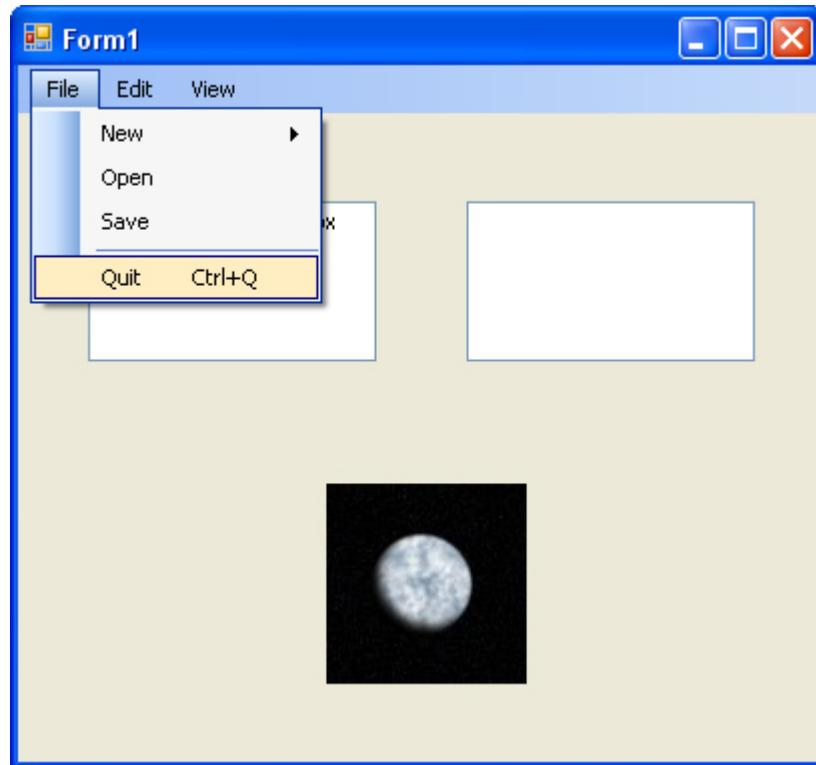
If **isNumber** is false, then you display an error message for your users. If it’s true, then it means that data from the text box can be converted properly. In which case, the rest of the code goes between the curly brackets of **else**.

If all that is a bit too complex then don’t worry about it – you’ll get there! In the next section, you’ll be doing something far easier than loops and Conditional Logic: we’ll show you how to add menus to your programmes.

But when you are first starting out, these are the two biggest hurdles to overcome: loops and Conditional Logic. When you understand these two difficult subjects then you are well on your way to becoming a programmer!

Adding Menus to Windows Forms in C#

In this section, we'll show how to add menus to your forms. You'll add File, Edit, and View menus, with items on each menu, and even sub menus. Here's what you will create:

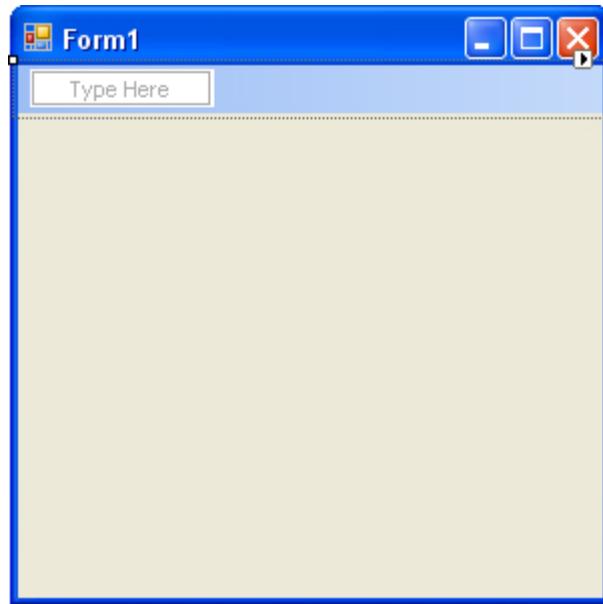


So start a new project by clicking **File > New Project** from the menu at the top of Visual Studio. Create a new Windows Application project. Call it anything you like. When your new form appears, you can add a menu bar quite easily.

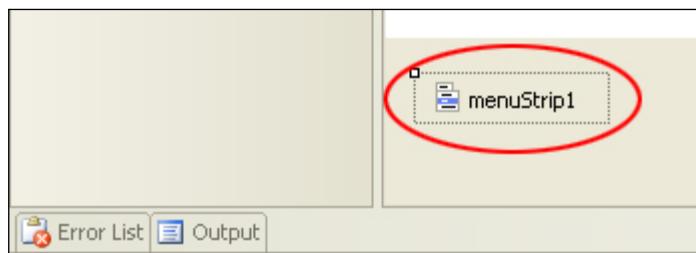
Have a look at the Toolbox on the left of Visual Studio. As well as the **Common Control** tools, there is a section for **Menus and Toolbars**. Click the plus or arrow symbol next to this to see the following:



The one you want is **MenuStrip**, which is highlighted in the image above. Double click MenuStrip and you'll see a menu bar appear at the top of your form:



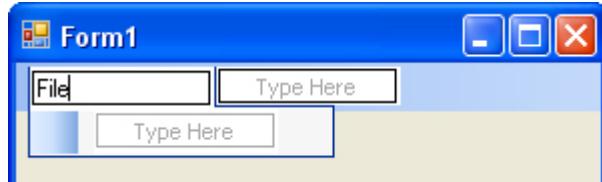
But notice what has appeared at the bottom of your Visual Studio window:



This is the MenuStrip object itself. The default Name for the MenuStrip is **menuStrip1**. If your MenuStrip is not selected, you can click on this icon at the

bottom. When you do, you'll see all the Properties for the MenuStrip appear in the Properties Window on the right hand side of Visual Studio.

Adding items to your menus is quite simple. Click inside of the area at the top, where it says "Type Here". Now type the word **File**.



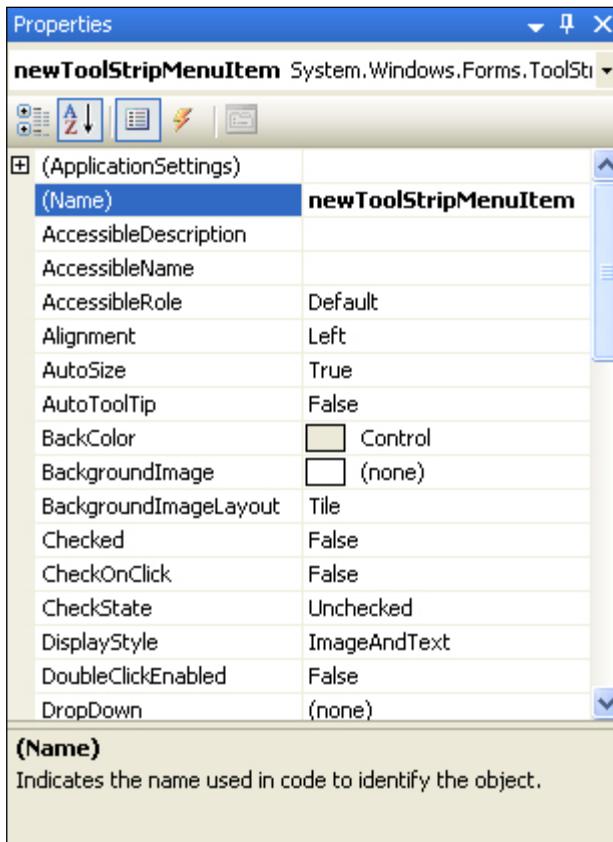
Hit the Enter key on your keyboard and your menu will look like this:



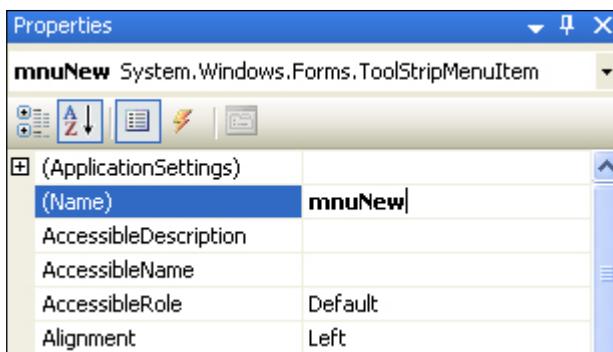
What you have done is to create the main menu item. To add items to your File menu, click inside of the second "Type Here" area pictured above. Now type the word **New**. Hit the Enter key on your keyboard to add the menu item:



Click back on the word "New" after you have hit the enter key. This will select just this menu item, and no other. Once you create a menu item it has its own Properties that you can change. With the New item selected, have a look at the Properties Window on the right hand side of Visual C#:

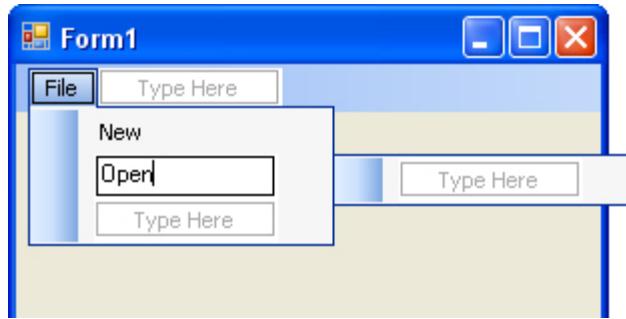


The Property we're interested in is the Name. It's a bit too long at the moment. So change it to **mnuNew**, as in the image below:



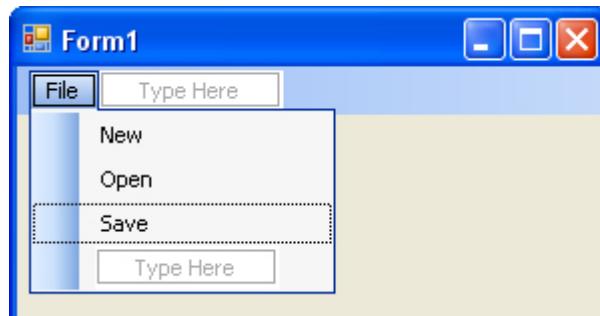
If you scroll down, you'll also see a Text property. It will say **New**, at the moment. It says "New" because that's what you typed in the menu bar when you created this item. You could change this here, if you wanted to. But leave the Text property on **New**.

Click back on your menu at the top of your form, and then into the "Type here" area just below **New**. Type the word Open:

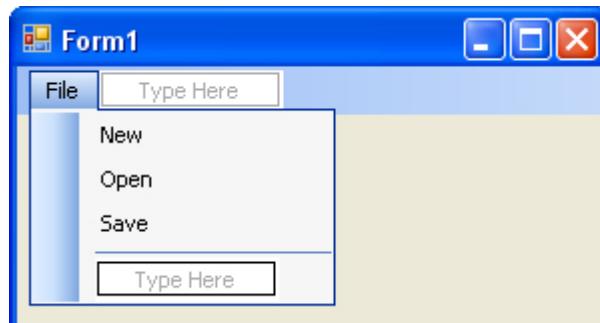


Hit the Enter key on your keyboard to create the **Open** menu item. Change its Name property, just like you did for the New item. Change the name to **mnuOpen**.

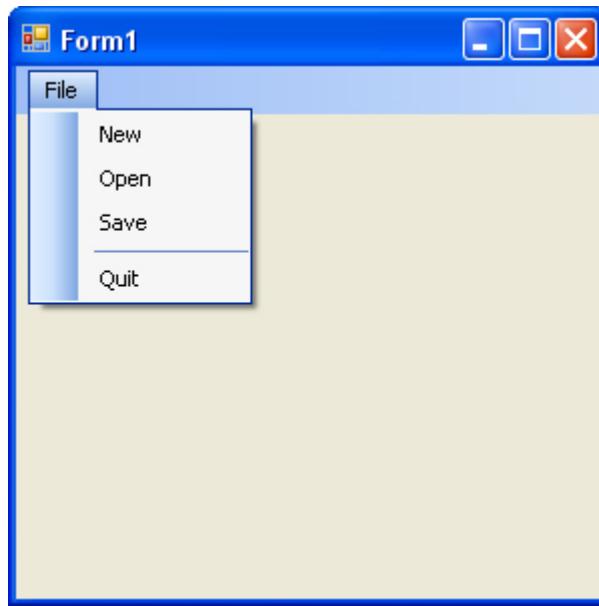
Create a **Save** menu item, underneath **Open**. Change its Name property to **mnuSave**. Your File menu will then look like ours below:



We'll now create just two more menu items, a dividing line, and a Quit item. To create a dividing line, click inside of the "Type Here" area below **Save**. Now type a hyphen (just to the right of the zero key on a UK keyboard). When you press the Enter key, C# will turn the hyphen into a dividing line. It should look like this:



Add the Quit item below your dividing line. Change the Name property to **mnuQuit**. Your File menu is now complete. To see what it looks like, run your programme. You should have a blue toolbar running across the top with a File menu. Click your File menu:

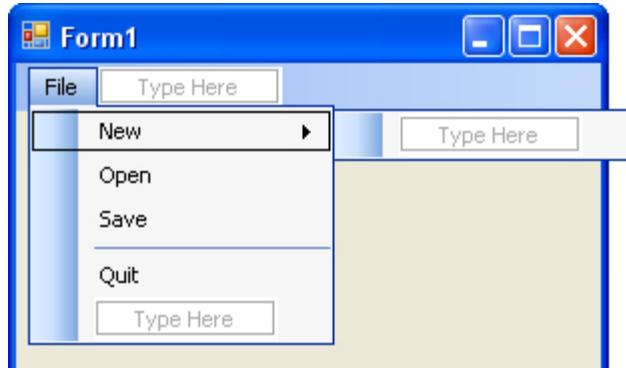


Of course, none of the menus work, because you haven't written any code for them yet. We'll do that soon.

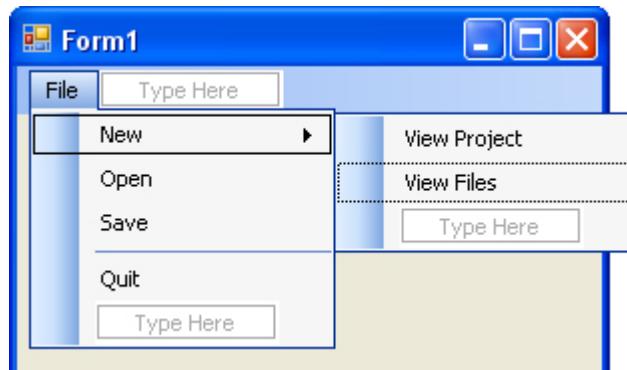
How to Add a Sub Menu

You can add Sub Menus just as easily. A Sub menu is one that opens out from a main menu item.

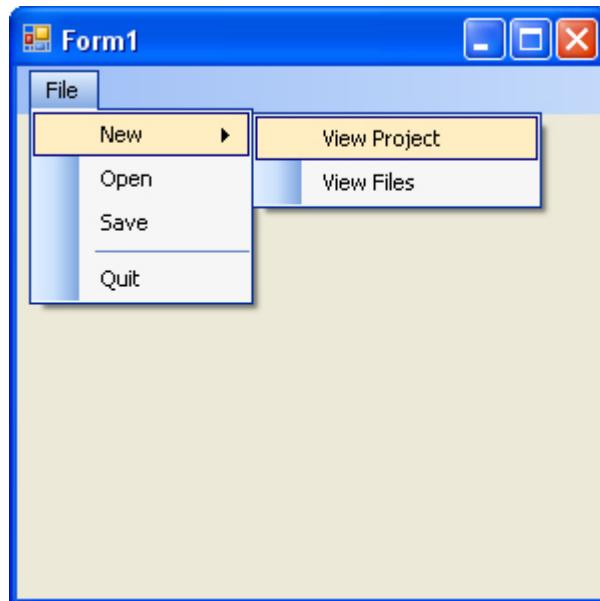
Halt your programme and return to your form. Click on the **New** item to select it. You should see a "Type Here" box appear to the right of **New**:



Click Inside of this box and type **View Project**. Hit the enter key and type **View Files** in the box below this. Your menu will then look like this:

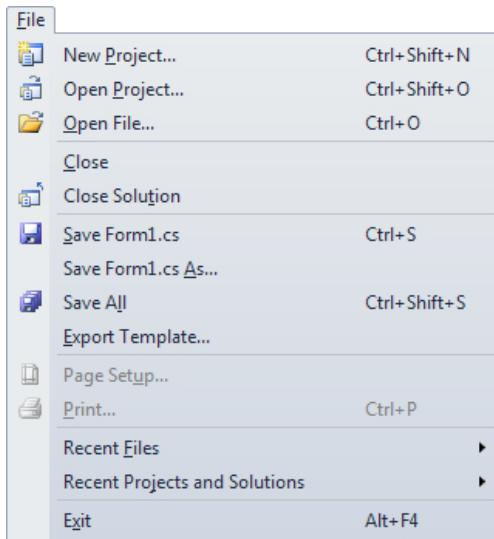


When the form is run, the Sub Menu will look like this:



Menu Shortcuts

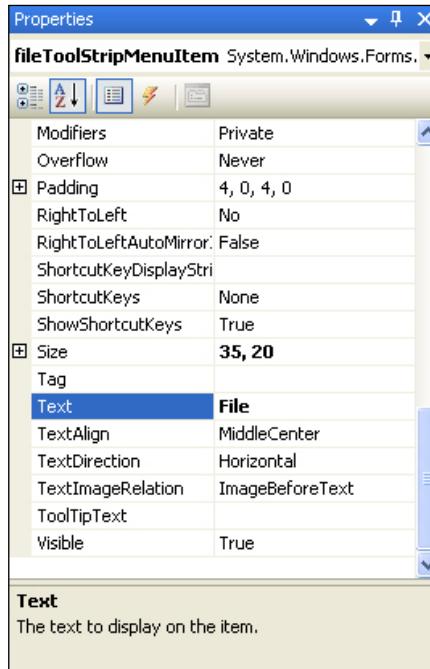
Menus usually have shortcuts. These are the underlined letters that you see when you click a menu. They sometimes have a shortcut key combination to the right of the menu item. For example, here's the File menu from Visual C# Express with all the underlines and key combinations showing:



To see these, you need to hit the ALT key on your keyboard. When you see the underlined letters, press the key that corresponds to the underlined letter. Pressing the “F” key, for example, will then cause the menu to drop down. Pressing any of the underlined letters on the File menu will implement that menu item. (In our edition of Visual Studio Express, pressing the letter P only switches back and forward between the first two items. The other letters work OK, though.)

You can also use the key combinations to the right of the menu item. Holding down CTRL + SHIFT + N at the same time will cause the New Project dialogue box to appear.

To add shortcuts to your own menus, click the **File** item to select it. Now have a look at its Properties in the Property Window. Scroll down until you locate the **Text** item:



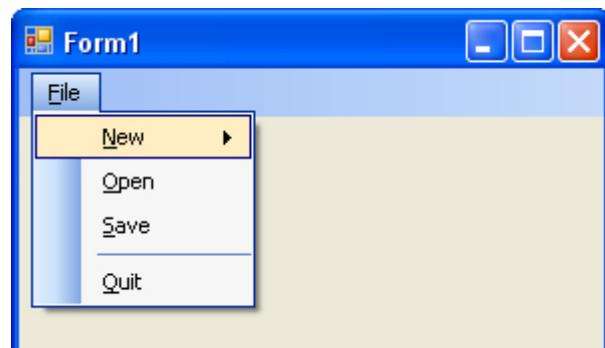
To add an underline to any of the letters, you use the ampersand symbol (&) before the letter you want to use as a shortcut. In the image below, we've added an ampersand just before the “F” of File:

| | |
|---------------|------------------|
| Tag | |
| Text | &File |
| TextAlign | MiddleCenter |
| TextDirection | Horizontal |

And here's what the menu looks like with the ampersand added:

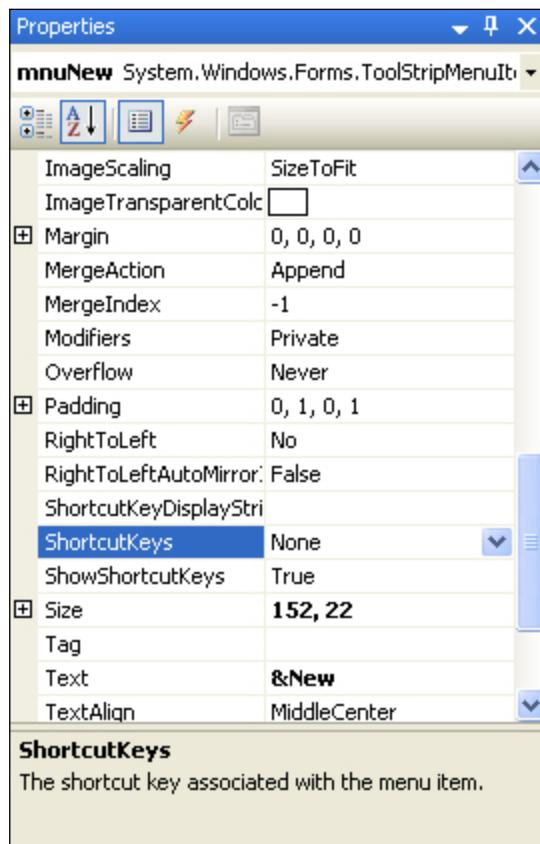


As you can see, there's now a line underneath the letter “F”. In the next image, we've added more underlines to the rest of the File menu:

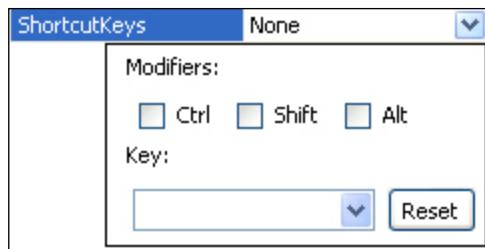


Add the same underlines to your own File menu. Remember: click a menu item to select it, locate the text property, and add an ampersand before the letter you want to use as a shortcut.

The key combination shortcuts are just as easy to add. Click on your **New** menu item to select it. Locate the **ShortcutKeys** Property in the Properties Window:

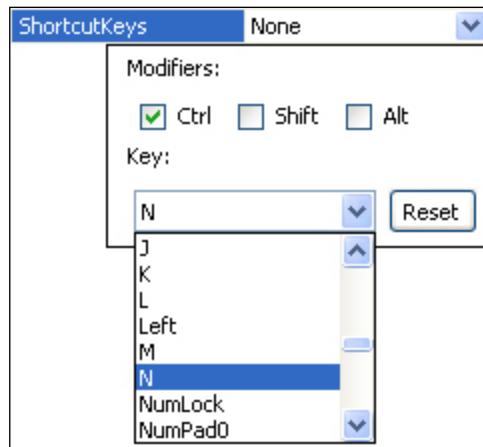


At the moment, it's set to **None**. Click the down arrow to see the following options:

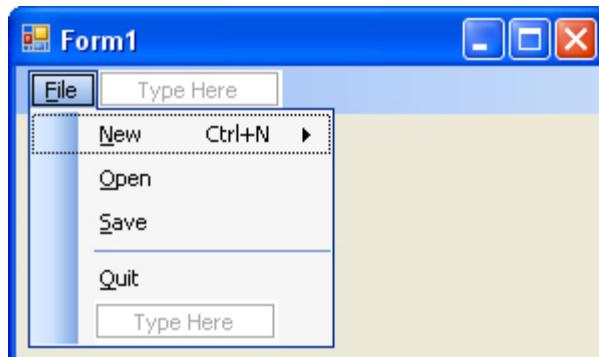


The Modifiers are the CTRL, Shift, and ALT keys. You can select one or all of these, if you want. To activate a shortcut, you would then have to hold down these keys first. So if you want your users to hold down the CTRL and Shift keys, plus a letter or symbol, then you would check the relevant Modifier boxes above.

The letters and symbols can be found on the Key drop down list. Click the down arrow to see the following:

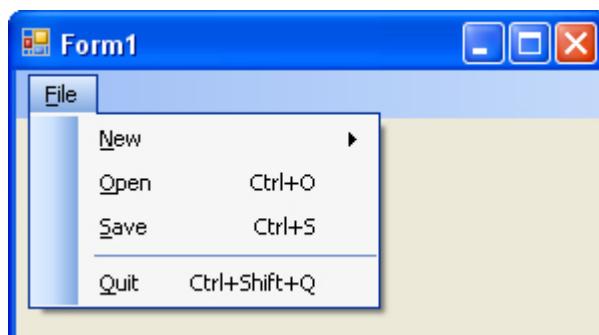


In the image above, we've gone for the CTRL modifier, and the letter "N". Clicking back on the menu, here's what it now looks like:



As you can see, the shortcuts for the New menu item are an Underline, and Ctrl + N.

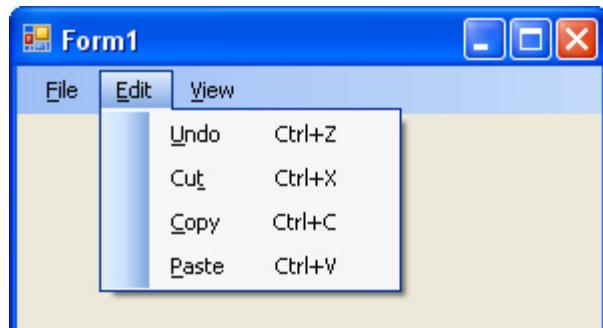
Have a look at the next image, and add the same Shortcut Keys to your File menu:



The ones you are adding are the final three: Open, Save and Quit. We'll get to coding the menu items shortly, but here's an exercise to complete. (Don't skip this exercise because you'll need the menu items!)

Exercise

Add an Edit menu to your menu bar with the following items:



Include the underline shortcuts, and the key combination shortcuts. For the Name Property of each menu item, use the following:

| | |
|--------------|-----------------|
| Undo | mnuUndo |
| Cut | mnuCut |
| Copy | mnuCopy |
| Paste | mnuPaste |

Exercise

Add a View menu to your menu bar with the following items:



Again, include the underline and key combination shortcuts. Set the Name Property of your menu items to the following:

| | |
|------------------------|-------------------------|
| View Text Boxes | mnuViewTextBoxes |
| View Labels | mnuViewLabels |
| View Images | mnuViewImages |

OK, it's now time to do some coding for the menu items you have created.

Adding code for your menu items

Of course, a menu is no good if nothing happens when you click an item. So we need to add code behind the menu items. We'll start with the Quit item, which should be on your File menu. There's only one line of code for this.

Return to your form, and click the menu strip. Click the File item to see its menu. Double click on your **Quit** item and the coding window should open. Your cursor should be flashing between the curly brackets of the Quit code stub:

```
private void mnuQuit_Click(object sender, EventArgs e)
{
    |
}
```

Notice that the Name you gave your menu item is used in the code stub: **mnuQuit**. But when a user clicks your Quit menu, you want the programme to end. To close down a Windows application, you can use this:

Application.Exit();

So add that line of code between the curly brackets of your Quit code stub. Run your programme and test it out. Hold down CTRL + SHIFT, and then the letter Q on your keyboard. The programme should close straight away. It does this because of the key combination shortcuts you added.

To see your underline shortcuts in action, start your programme again. Press the ALT key on your keyboard and you should see all the underlines appear for File, Edit and View. Press the letter “F” on your keyboard (the underlined letter), and the menu should drop down. Now press the letter Q on your keyboard. Because this is the underlined letter, the programme should exit.

You can add more code to menu items – anything you like, in fact. Something you see on Quit menus is a message box:

Are you sure you want to Quit?

To add a message box to your own code, try this:

```
if (MessageBox.Show("Really Quit?", "Exit",
    MessageBoxButtons.OKCancel) == DialogResult.OK)
{
    Application.Exit();
}
```

The first two lines should be one line in your code. It's only on two lines here because there's not enough room for it on this page. But in between the round brackets of an if statement, we have a message box:

`MessageBox.Show("Really Quit?", "Exit", MessageBoxButtons.OKCancel)`

This will get you a dialogue box with OK and Cancel buttons. C# will wait until the user clicks a button. Our code uses an if statement to test which button was clicked. To see which one the user clicked, you add this on the end:

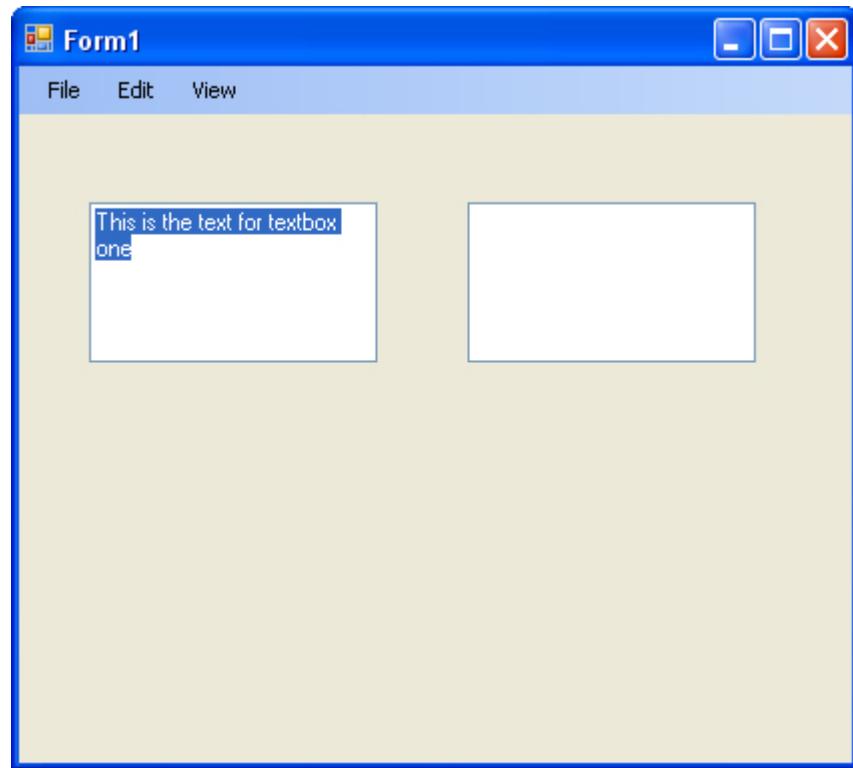
```
= = DialogResult.OK
```

So the line reads “IF the result of the dialogue box was that the OK button was clicked, then Exit the Application.”

The Edit menu

We’ll leave the rest of the File menu till last, as it’s a bit more complicated. Our Edit menu is not too difficult, as there’s only one line of code for each menu item.

To see Cut, Copy, Paste and Undo in action, add two text boxes to your form. Set the **MultiLine** Property of each text box to **true**. This will allow you to have a text box with more than one line of text. For text box one, type anything you like for the Text property. Your form should then look like this:



What’s we’ll do now is to enable the Cut, Copy and Paste menu items, as well as the undo. We’ll first **Cut** the highlighted text from the text box and then **Undo** the operation.

So return to your Form. Click your blue menu bar at the top, and click on your Edit menu. Double click the Cut item. C# will create a code stub for you. If you

gave your Cut menu item the Name of **mnuCut** then your code stub will look like this:

```
private void mnuCut_Click(object sender, EventArgs e)
{
    |
}
```

The code to cut any highlighted text is quite simple. Add this between the curly brackets of your **mnuCut** code:

```
textBox1.Cut();
```

Cut() is a method that's built in to C#. It works on text boxes, amongst other things, and does what it says – cuts text!

Before you try it out, return to your form and double click your Undo menu item. Add the following line for the code:

```
textBox1.Undo();
```

Now try it out. Run your form and highlight the text in the text box. Use your Cut menu to cut the text. Then use your Undo menu to restore the text.

You can also check to see if any text was selected. Change your code to this:

```
if(textBox1.SelectedText != "")
{
    textBox1.Cut();
}
```

We're using an if statement to check a property of text boxes called **SelectedText**. This can tell you if any text is selected. We're using the “Does Not Equal” operators (!=) followed by a pair of double quotes. A pair of double quotes with no spaces means that it's a blank string of text. If text was selected, then the if statement is true. In which case the Cut operation can go ahead.

You can manipulate selected text with the **SelectedText** Property. In the code below, we're handing the selected text to a string variable and displaying it in a message box:

```
string someText;

if(textBox1.SelectedText != "")
{
    someText = textBox1.SelectedText;
    MessageBox.Show(someText);
}
```

For the Undo menu, you might want to check if the operation can actually be Undone. If you want to check for this, there is another property of text boxes called **CanUndo**. You use it like this:

```
if (textBox1.CanUndo == true)
{
    textBox1.Undo();
}
```

Only if the operation can be Undone will the code for the if statement execute.

However, if you run your programme and cut some text, clicking Undo twice will first restore the text and then cut it again! (You're undoing the restore.) To remedy this, you can clear the undo operation. Change your code to this. The new line is in bold text below:

```
if (textBox1.CanUndo == true)
{
    textBox1.Undo();
textBox1.ClearUndo();
}
```

So you just add the **ClearUndo()** method after the dot of **textBox1**.

Try it again and you'll find that clicking Undo twice won't cut the text again.

Copy and Paste

To Copy something to the Clipboard, you highlight text and click a Copy item on an Edit menu. Once the data is copied to the Clipboard, it can be Pasted elsewhere. We'll implement this with our menu system.

Double click the Copy item on your Edit menu. You'll be taken to the code stub for your Copy menu item. Add the following code between the curly brackets:

```
textBox1.Copy();
```

Using the Copy method of text boxes is enough to copy the data on to the Windows Clipboard. But you can first check to see if there is any highlighted text to copy. Change your code to this:

```
if (textBox1.SelectionLength > 0)
{
    textBox1.Copy();
}
```

We're using an if statement again. This time, we are checking the **SelectionLength** property of text boxes. The length returns how many characters are in the text that was selected. We want to make sure that it's greater than zero.

We'll use the second text box to Paste. So access the code stub for your Paste menu item, using the same technique as before. Add the following between the curly brackets of your Paste code:

```
textBox2.Paste( );
```

Notice that we're now using **textBox2** and not **textBox1**. After the dot, you only need to add the Paste method.

Try your Edit menu out again. Highlight the text in the first text box. Click **Edit > Copy**. Now click into your second text box and click **Edit > Paste**.

You can also check to see if there is any data on the Clipboard, and that it is text and not, say, an image. Add this rather long if statement to your code:

```
if (Clipboard.GetDataObject().GetDataPresent(DataFormats.Text) == true)
{
    textBox2.Paste( );
    Clipboard.Clear( );
}
```

Getting at the data on the Clipboard can be tricky, but we're checking to see what the DataFormat is. If it's text then the if statement is **true**, and the code gets executed. Notice the last line, though:

```
Clipboard.Clear( );
```

As you'd expect, this Clears whatever is on the Clipboard. You don't need this line, however, so you can delete it if you prefer. See what it does both with and without the line.

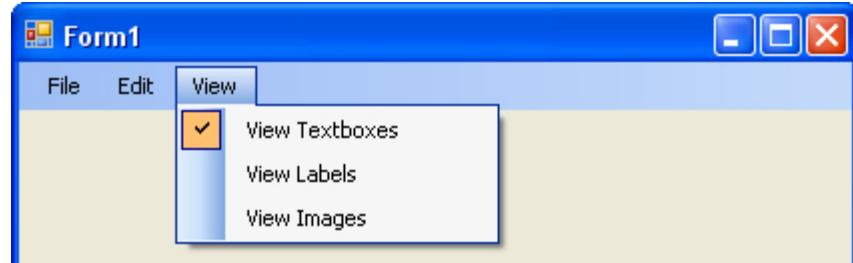
The View Menu

We have three items on our View menu. But we'll only implement two of them. For the first one, View Text Boxes, we'll show you a handy programming technique with Boolean variables – how to toggle them on and off.

So return to your form, and double click the menu item for View Text Boxes. C# will generate the code stub for you:

| |
|---|
| <pre>private void mnuViewTextboxes_Click(object sender, EventArgs e) { }</pre> |
|---|

What we'll do is to hide the text boxes when the menu item is clicked, and unhide the text boxes when you click again. A check icon will then appear or disappear next to the menu item. Here's an image of what we'll be doing:



To place a check mark next to a menu item, you use the **Checked** Property of the menu item. Add this to your View Textboxes code stub, in between the curly brackets:

```
mnuViewTextboxes.Checked = true;
```

So you just type a dot after the Name of your menu item. Then select the **Checked** property from the IntelliSense list. Checked is a Boolean value that you either set to true or false (it's either got a check mark next to it or it hasn't).

Run your programme and click your View Textboxes menu item. You should see a check appear. It does so because the default value for the Checked property is false. It only becomes true when you click the menu item, thereby running the code you added.

The question is, how do you get the Check mark symbol to disappear when it's clicked again? Obviously you need to set it to **false**, meaning not checked. But what's the code?

A handy programming technique is to toggle Boolean values off and on. You do it with the aid of the NOT operator (!). Amend your code to this:

```
mnuViewTextboxes.Checked = !mnuViewTextboxes.Checked;
```

So instead of setting the Checked value to true, we have this:

```
!mnuViewTextboxes.Checked;
```

This says, "NOT Checked". But doesn't mean "Unchecked". What you are doing is setting the Boolean variable to what it is currently NOT. Remember: Checked can either be true OR false. So if Checked is currently true, set it to false, and vice versa. The result then gets stored back in the Property on the left of the equals sign.

Run your programme and try it out. Click the menu item to see the Check symbol. Click it again and it will disappear. This toggling of Boolean variables is quite common in programming, and can save you a lot of tricky coding!

To actually do something with the text boxes, though, you can add an if statement to examine whether the variable is true. What we'll do is make the text boxes visible if there's a Check, and not visible if there isn't a Check. Add this code just below the line you already have:

```
if (mnuViewTextboxes.Checked)
{
    textBox1.Visible = true;
    textBox2.Visible = true;
}
else
{
    textBox1.Visible = false;
    textBox2.Visible = false;
}
```

The Property we are changing is the **Visible** Property of text boxes. As its name suggests, this hides or un-hides an object. Again, it's a Boolean value, though. So we could have just done this:

textBox1.Visible = !textBox1.Visible;

The use of the NOT operator will then toggle the Visibility on or off. We added an if statement because it's handy to actually examine what is in the variable, rather than just assuming.

One line you may puzzle over is this:

if (mnuViewTextboxes.Checked)

The part in round brackets could have been written like this, instead:

if (mnuViewTextboxes.Checked == true)

For if statements, C# is trying to work out if the code in round brackets is **true**. So you can leave off the “**= true**” part, because it's not needed. If you want to check for false values, you can use the NOT operator again. Like this:

if (!mnuViewTextboxes.Checked)

This is the same as saying this:

if (mnuViewTextboxes.Checked == false)

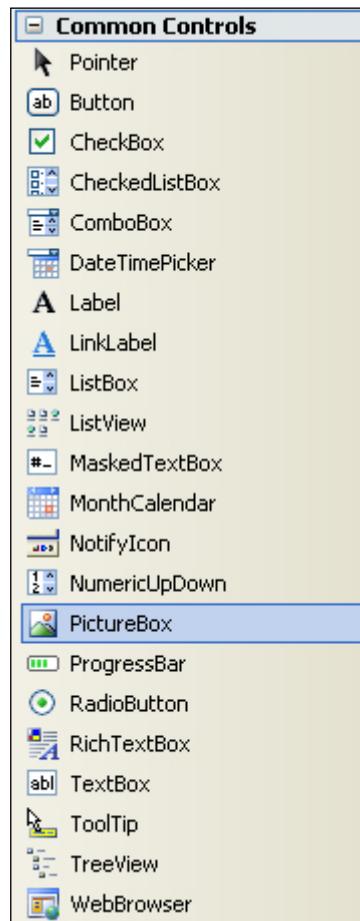
Using the NOT operator is considered more professional. They mean the same, though, so use which one is better for you.

View Images

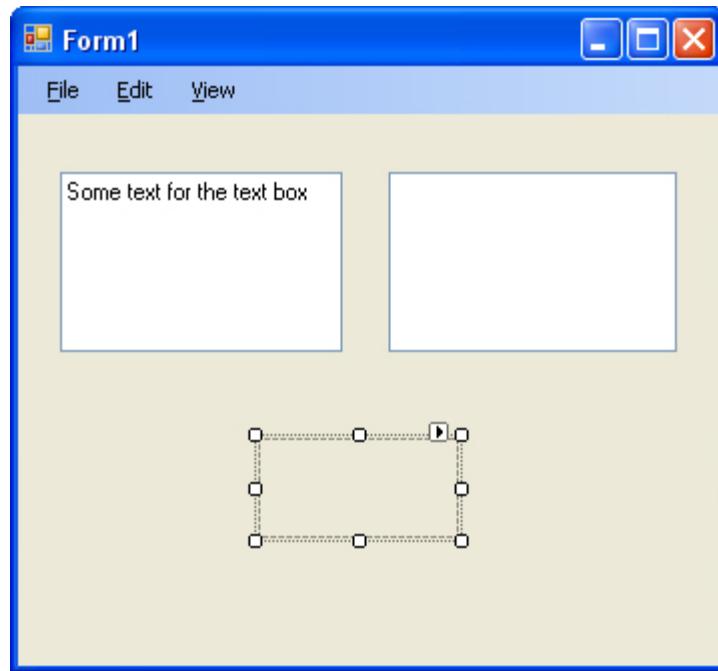
You have seen the Open File dialogue box countless times. It's the one that appears whenever you click **File > Open** on a Windows machine. You then navigate through folders, searching for the file you want to open. For our **View Images** menu, we'll do something slightly more complex – we'll have our own Open File dialogue box that allows you to select images from your computer. When you select an image, it will then appear in a new control on your form.

So we need a place on our form where we can store images. We'll use a Picture Box.

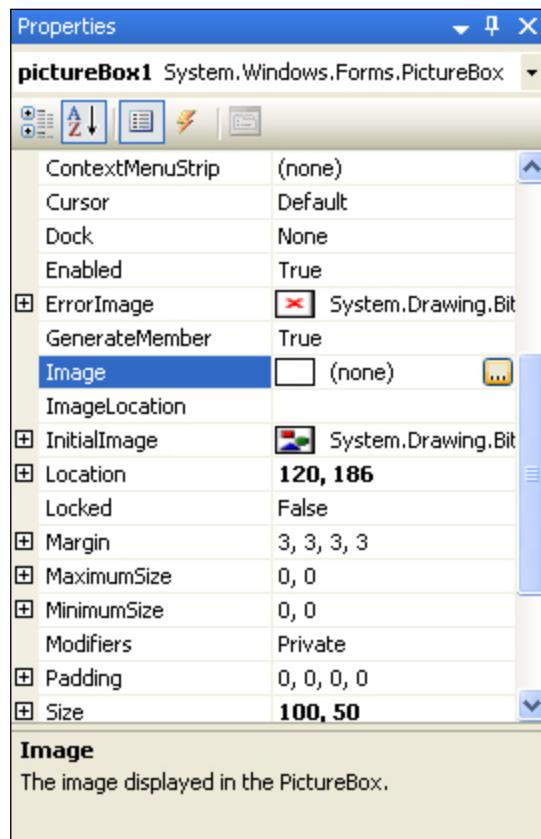
Have a look at the Toolbox on the left hand side of Visual Studio. Under Common Controls, locate PictureBox:



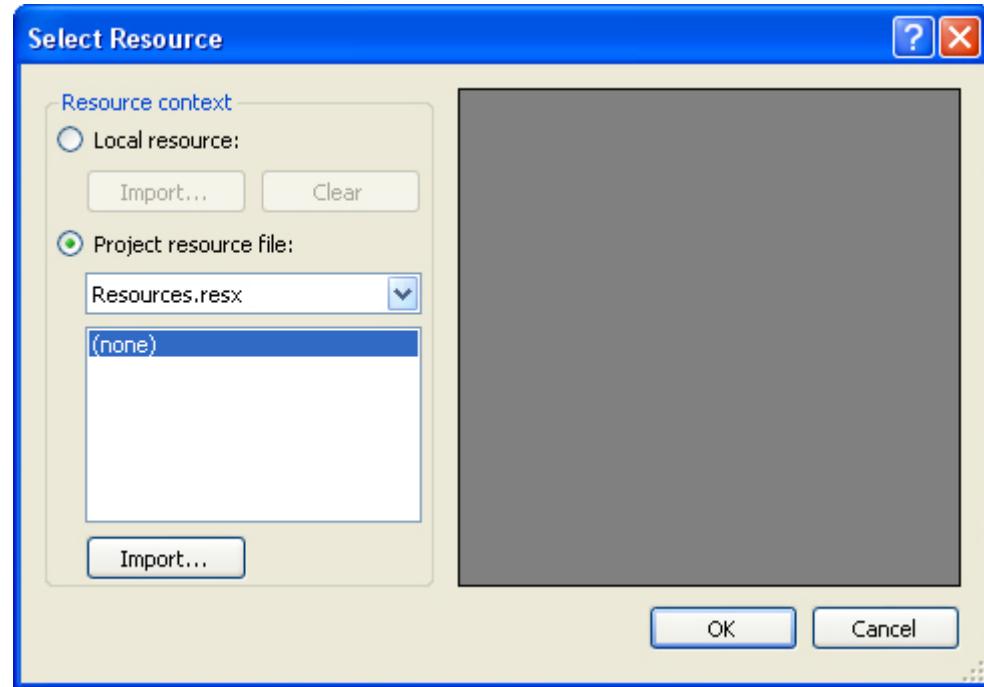
Once you've select the PictureBox tool. Click on your form once to add a new PictureBox control. Your form should then look like this:



The PictureBox control is blank when you first add one. To add an image to it at Design Time, have a look at the Property Window. Locate the **Image** property:

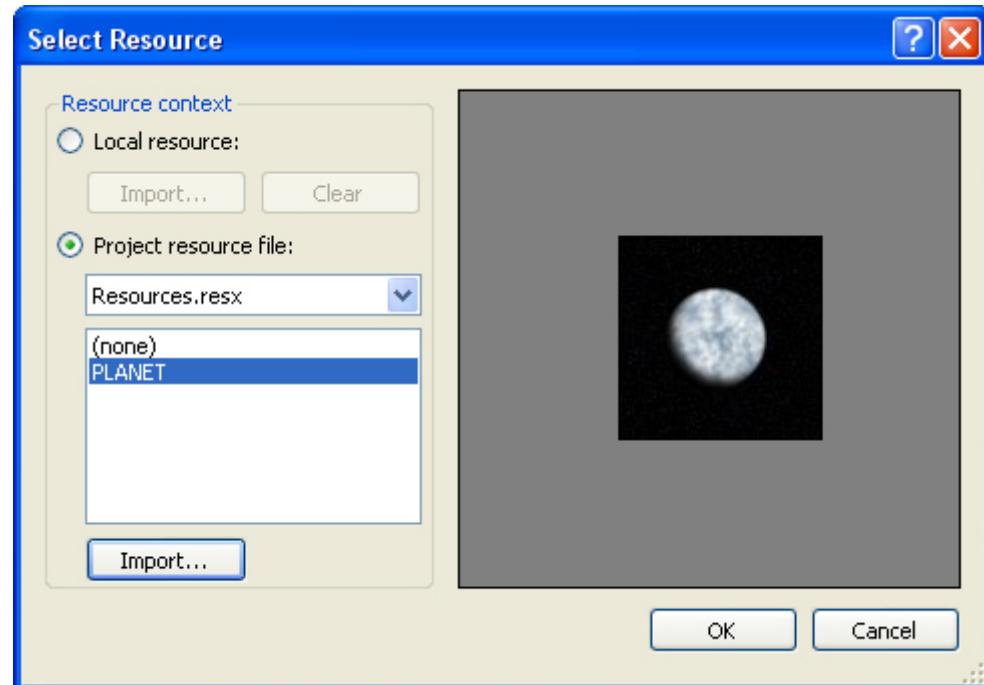


Click the button with the three dots on it to see a dialogue box appear:

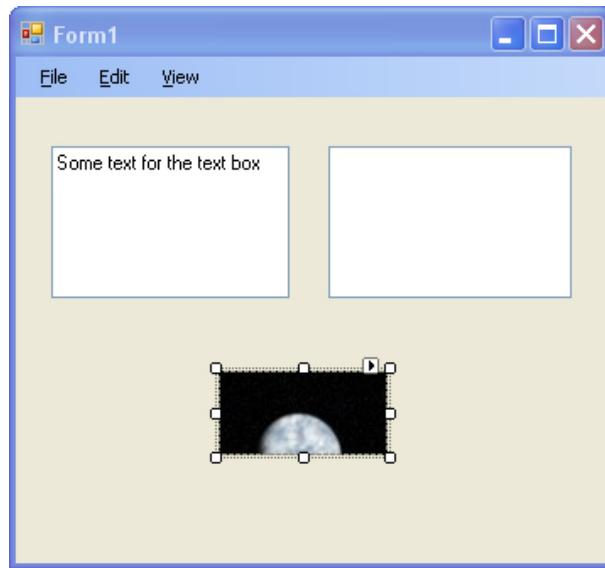


Click the **Import** button at the bottom and you'll see a standard Open dialogue box. Search your hard drive for a suitable image. Because you have "Project resource file" selected, C# will copy the image to a folder in your project. (This is handy if you want to send your programme to anyone else.)

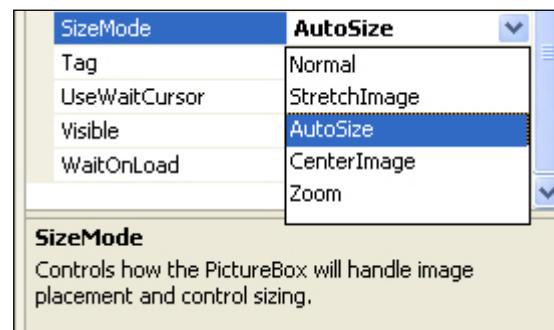
In the image below, we've gone for a picture of a planet:



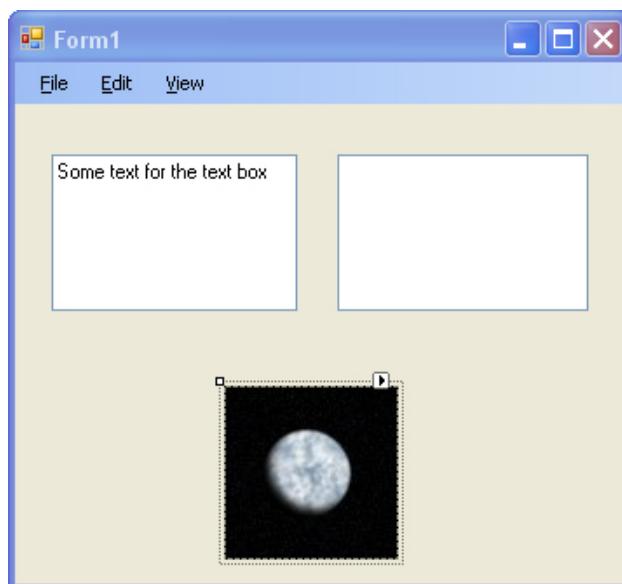
Click OK and you'll be taken back to your form:



Notice that the image is too big for the PictureBox. Locate the **SizeMode** property in the Properties Window:



As you can see, there are a few to choose from. Select **AutoSize**, and the PictureBox will automatically stretch to the size of your image:

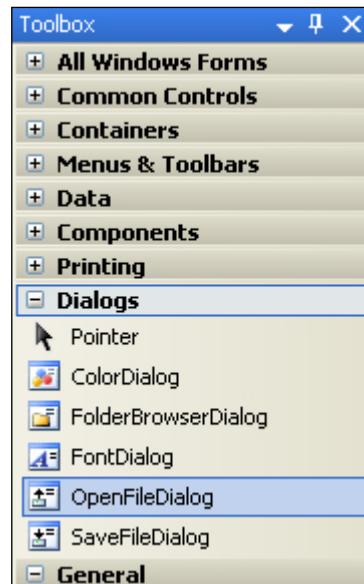


If you run your programme, you'll see the image appear on the form. It won't have a border, though. If you want a border, explore the **BorderStyle** Property of your PictureBox control.

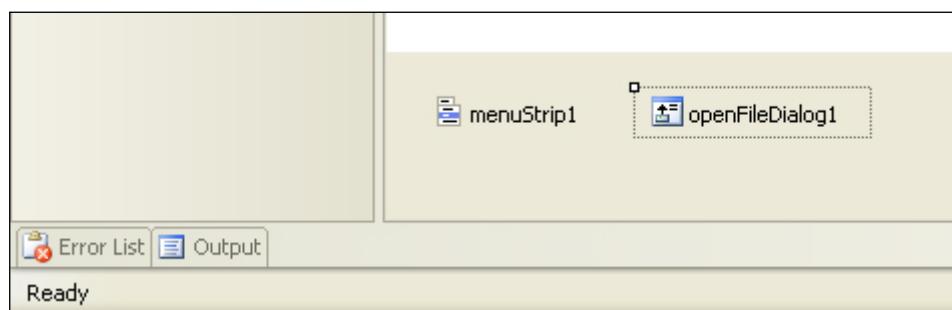
The Code to View and Insert Images in C#

We'll now give users the option to add their own images to the picture box, instead of the one we chose. To do that, you need to display an Open File dialogue box when the user clicks your **View > View Images** menu item.

Dialogue boxes in C# can be added with the aid of an inbuilt object. Have a look in the Toolbox on the left hand side of Visual Studio. There should be a category called **Dialogs**:

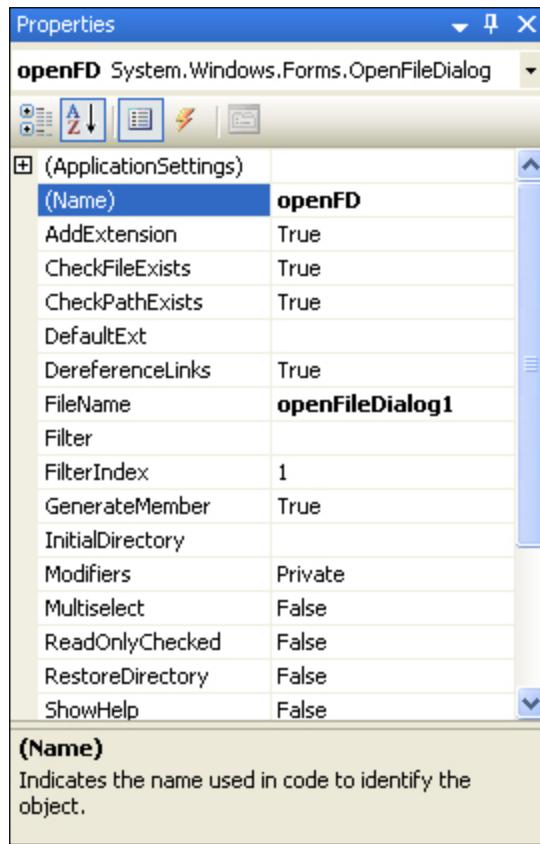


All the dialogue boxes you are familiar with in Windows are on the list above. The one highlighted is the one we want – **OpenFileDialog**. Double click this item, and you'll see a new item appear at the bottom of Visual Studio, next to your menuStrip1 object:

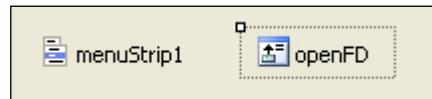


Nothing will appear on your form, however, because the Dialog controls are hidden from view. The one in the image above has a default Name of

openFileDialog1. This is a bit long, so have a look at the Properties Window on the right. Change the Name to **openFD**:



The control at the bottom of Visual Studio should have changed, as well:



With the control selected, have another look at the Properties Window. You'll see that there are Properties for Filter, FileName, InitialDirectory and Title. We'll change these with code. But one important point to bear in mind about the Open File Dialogue box is this: They don't actually open files! What the Open File Dialogue box does, and the same is true for the other Dialog controls, is to allow you to select a file for opening. You have to write separate code to open anything. The only thing you're really doing here is to get at a file name.

We want the dialogue box to appear when the **View > View Images** menu is clicked. So double click this item on your **View** menu. A code stub will appear:

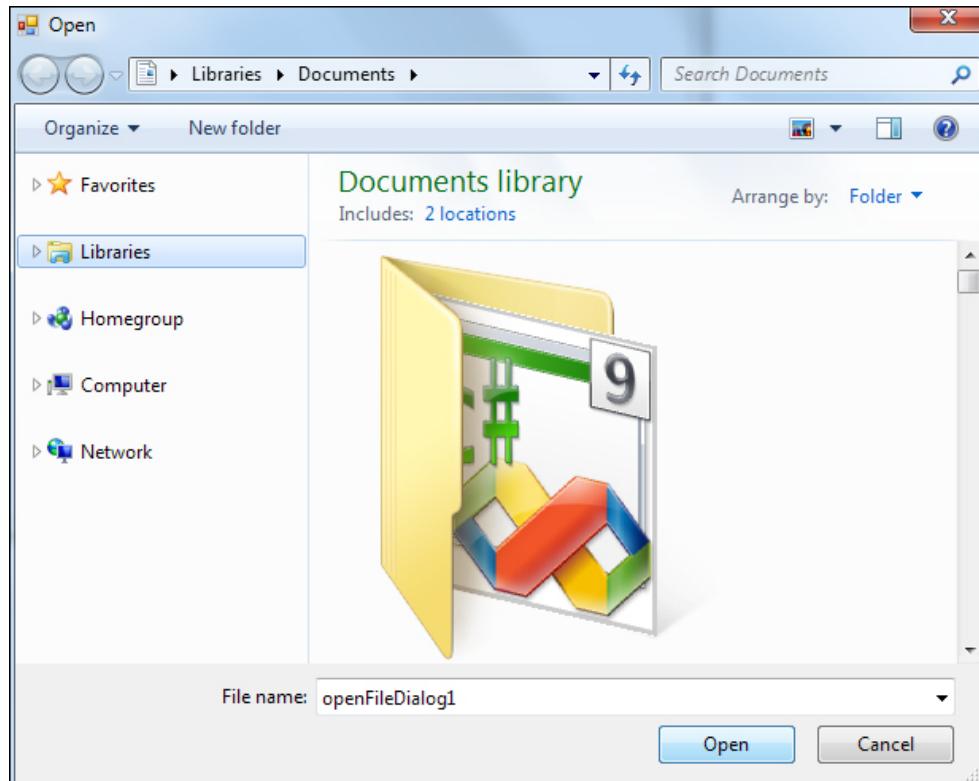
```
private void mnuViewImages_Click(object sender, EventArgs e)
{
}
```

To see the Open Dialogue box, add this line to your code, in between the curly brackets:

```
openFD.ShowDialog();
```

So you type the Name of your control, and then a dot. After the dot, select **ShowDialog** from the IntelliSense list. As its name suggests, this shows you the dialogue box.

Run your programme and try it out. You should see something like the following appear when you click your **View > View Images** menu item:

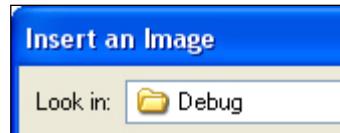


Because we haven't yet set any Properties, a default location is displayed, which is the Documents folder in Windows 7. The **File name** has the default `openFileDialog1`. You can change all these, though.

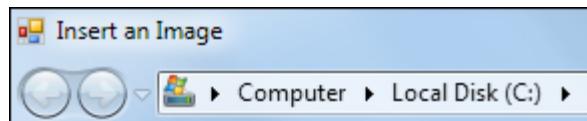
We can set a Title, first. The default Title is the word Open, in white on a blue background in XP, black on light blue background in Vista and Windows 7. Add this line to your code, before the first line:

```
openFD.Title = "Insert an Image";
```

This time, we're using the Title Property, and setting it to the text "Insert an Image". You can, of course, type anything you like here. When you run your programme and click the menu item, the new Title will look like this in XP:



And this in later versions of Windows:



Another thing you can change is that **Look in** area. You can reset it with the **InitialDirectory** property. Add the following line to your code, before the other two lines:

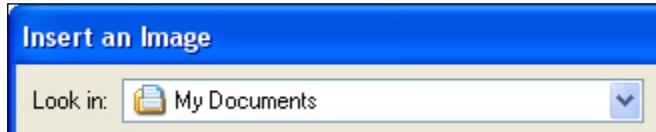
```
openFD.InitialDirectory = "C:";
```

We're setting the default folder to be C. This would assume that the user had a hard drive called C. If you want to set the Initial Directory to the “My Documents” folder of any computer, try this after the equals sign, instead of “C:”:

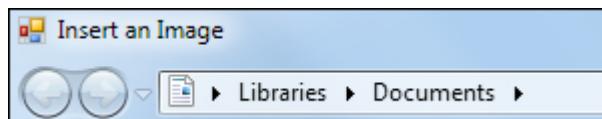
```
= System.Environment.GetFolderPath(Environment.SpecialFolder.Personal);
```

This will get the folder path to the My Document folder (Personal folder), which is called the Documents folder in Vista and Windows 7/8. You need to do it this way because different users will have different user names, and there's no way for you to tell beforehand.

But run your programme and try it out. The Look in box should have changed (XP):



Or this, in Windows 7/8:



For the File name area, you can use the **FileName** Property. Add this line to your code (add it before the final line):

```
openFD.FileName = "";
```

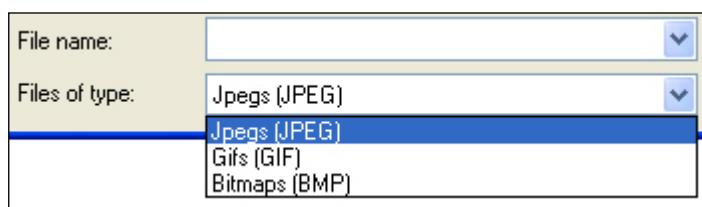
Here, we're setting the File Name to a blank string. Run your programme and you'll find that the **File name** area on your dialogue box will be blank, and the

cursor will be flashing away. Select any file you like, and the file name will appear in the box. But your code should now look like ours below:

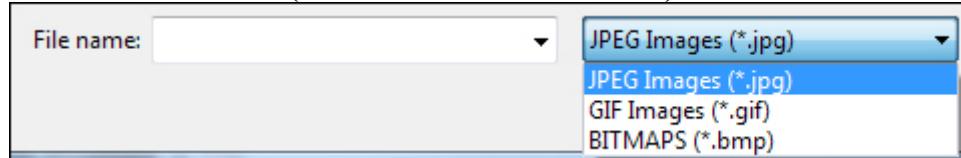
```
private void mnuViewImages_Click(object sender, EventArgs e)
{
    openFD.InitialDirectory = "C:";
    openFD.Title = "Insert an Image";
    openFD.FileName = "";

    openFD.ShowDialog();
}
```

The next thing to do is to set up some **Files of type**. This is for the drop down list you see at the bottom, just under **File name**. Here's what we want to do (XP):



(Later versions of Windows):



So we want the user to be able to select JPEG images, GIF images, and Bitmap images. When you set the files of type, you are restricting the type of files that the user can open. This is done with the **Filter** Property. After all, you don't want your users trying to insert text files into a picture box!

The filter property makes use of the pipe character (|). The pipe character can be found above the backslash on a UK keyboard. Add this code, just before the last line:

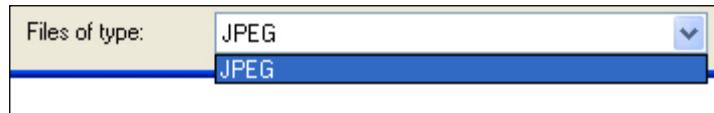
```
openFD.Filter = "JPEG|*.jpg";
```

Notice what comes after the equals sign:

```
"JPEG|*.jpg";
```

Your filters need to go between quote marks. But the JPEG part, before the pipe character, is what you want to display in the drop down list. You can have anything you like here, “JPEG Images” instead of just “JPEG”, for example. After the pipe character, you need an asterisk symbol * followed by a dot. The asterisk symbol means “any file name”. After the dot, you type the file extension that you want to filter for.

Run your code and try it out. You should see this in the “Files of type” list (on the right of the text box in Vista and Windows 7):



Now change your code to this:

```
openFD.Filter = "JPEG Images|*.jpg";
```

The “Files of type” list will then look like this, depending on your Operating System:



Later versions of Windows:



Using just one filter means that no other file types will display. To add other file types you just need to use the pipe character again. Let’s add GIF images, as well. Change your code to this:

```
openFD.Filter = "JPEG Images|*.jpg|GIF Images|*.gif";
```

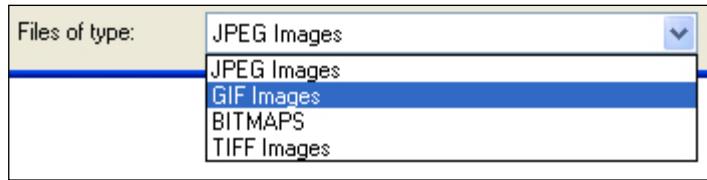
As you can see, the line is a bit messy! The new part is in bold, though. Notice that you separate one file type from another with a pipe character. But you also need a pipe to separate the text for the drop down list from the actual file type. To add Bitmap images, the code would be this:

```
openFD.Filter = "JPEG Images|*.jpg|GIF Images|*.gif|BITMAPS|*.bmp";
```

Here’s a few more image types, and their file extensions:

| | | | |
|-------------|-------|----|--------|
| TIFF Images | *.tif | or | *.tiff |
| PNG Images | *.png | | |
| PICT Images | *.pct | or | *.pict |

There are, of course, lots of others. In the image below, we’ve added TIFF files to the list. (Note that you can use upper or lower case for the extensions.):



To display files of any type, use an asterisk symbol in place of the file extension. For example:

```
openFD.Filter = "JPEG Images|*.jpg|All Files|*.*";
```

However, we still haven't inserted a new image. To place a selected image into the picture box, you have to get the file name that the user selected. You can add a string variable to your code for this:

```
string Chosen_File = "";
```

You then access the `FileName` property of `openFD`. Like this:

```
Chosen_File = openFD.FileName;
```

The file name will then be in the variable we've called **Chosen_File**.

To place a new image into the picture box you have on the form, you need the `Image` property:

```
pictureBox1.Image
```

To place your chosen file into the `Image` property, you need this:

```
pictureBox1.Image = Image.FromFile(Chosen_File);
```

So after the equals sign, you can use the `Image` object. This has a method called `FromFile()`. In between the round brackets of this method, you type the name of the image file. For us, this image file is stored in our `Chosen_File` variable.

Add the new lines to your code and your coding window should look something like ours below (we've cut down on a few filters):

```

private void mnuViewImages_Click(object sender, EventArgs e)
{
    string Chosen_File = "";

    openFD.InitialDirectory = "C:";
    openFD.Title = "Insert an Image";
    openFD.FileName = "";
    openFD.Filter = "JPEG Images|*.jpg|GIF Images|*.gif";

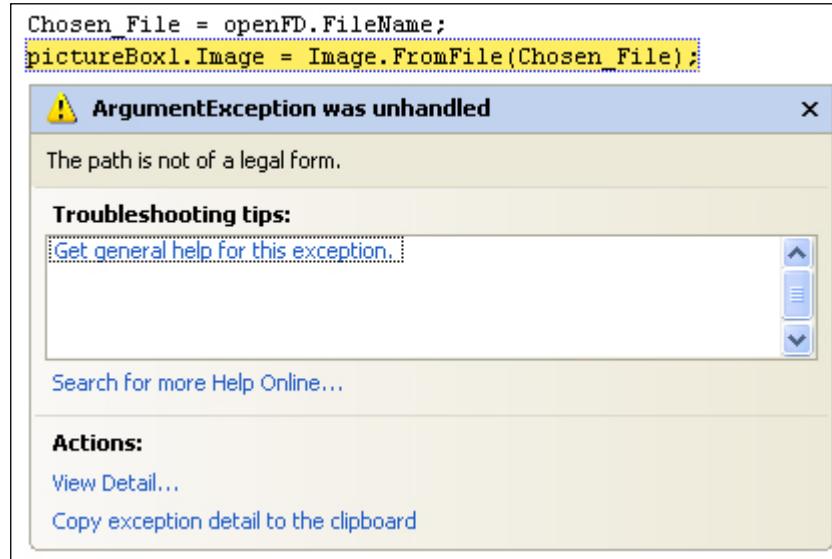
    openFD.ShowDialog();

    Chosen_File = openFD.FileName;
    pictureBox1.Image = Image.FromFile(Chosen_File);
}

```

Run your programme and test it out. Select an image to open. You should find that your new image replaces the old one in your picture box.

However, there is a problem with the code. Instead of clicking Open, click Cancel. You should get an error message (C# 2012's error message is a plain version of the one below):



Because the Cancel button was clicked, there is no image name in the variable Chosen_File. So the programme “bugs out” on you. You need to handle this in your code.

To check if the cancel button was clicked, you can use this:

```

if(openFD.ShowDialog() == DialogResult.Cancel)
{
    MessageBox.Show("Operation Cancelled");
}

```

So there is inbuilt object called **DialogResult**. You check if this has a value of Cancel. Adding an else statement gives us this code:

```

private void mnuViewImages_Click(object sender, EventArgs e)
{
    string Chosen_File = "";

    openFD.InitialDirectory = "C:";
    openFD.Title = "Insert an Image";
    openFD.FileName = "";
    openFD.Filter = "JPEG Images (*.jpg|GIF Images (*.gif)";

    if (openFD.ShowDialog() == DialogResult.Cancel)
    {
        MessageBox.Show("Operation Cancelled");
    }
    else
    {
        Chosen_File = openFD.FileName;
        pictureBox1.Image = Image.FromFile(Chosen_File);
    }
}

```

Change your code so that it looks like ours above. When you run your programme now, it shouldn't crash when you click the Cancel button.

You can also have this for you IF Statement, instead of the one above:

```

if (openFD.ShowDialog() != DialogResult.Cancel)
{
    Chosen_File = openFD.FileName;
    pictureBox1.Image = Image.FromFile(Chosen_File);
}

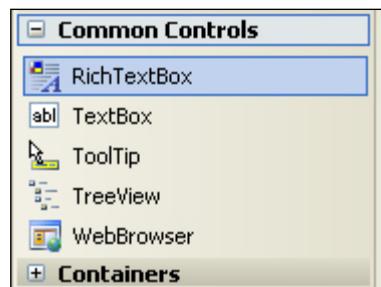
```

We've used the NOT symbol, here (!). So we're checking if **DialogResult** does NOT equal Cancel.

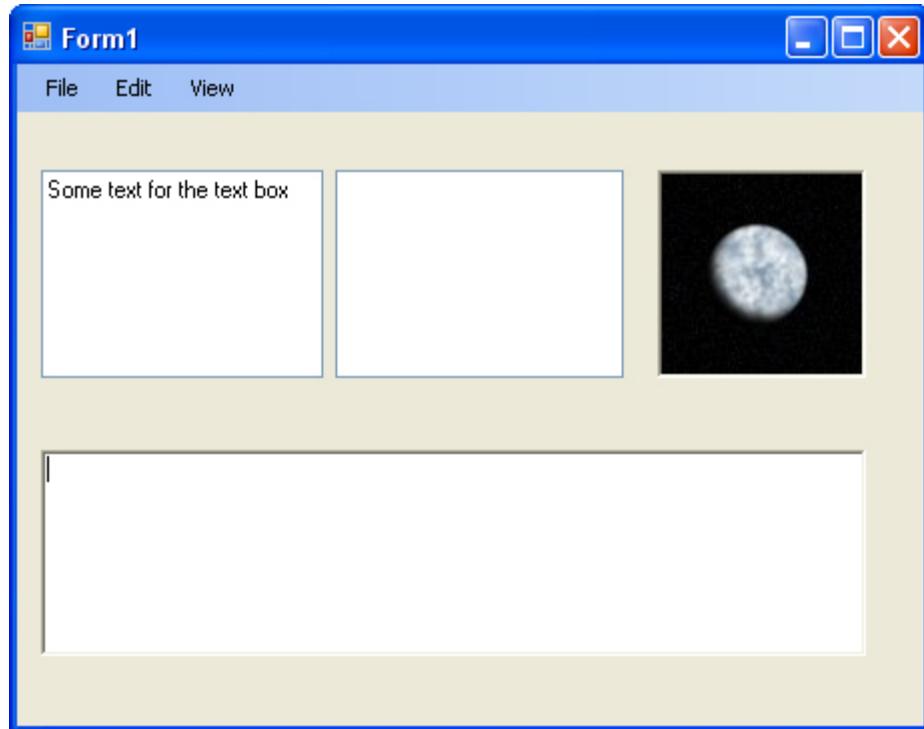
The Open File Dialogue Box

We can reuse the Open File dialogue box that we have added. Instead of filtering for images, we'll filter for text files. We'll also add a different kind of text box – the Rich Text Box. This will allow us to easily add the text from the file, straight into the programme.

So return to Designer View, so that you can see your form. Now expand the Toolbox, and locate **RichTextBox**, under Common Controls:



Double click to add a RichTextBox to your form. You may have to adjust the height and width of your form, and reposition other controls. But your form should look like this, when you've add the RichTextBox:



The RichTextBox is the one at the bottom – it looks exactly the same as a normal text box, but you can do more with it. One Method it does have is called LoadFile(). We'll use this to load a text file.

Now that we've added the RichTextBox, we can add some code. So, access the code stub for you **File > Open** menu item. It should look like this:

```
private void mnuOpen_Click(object sender, EventArgs e)
{
    |
}
```

We can add the same lines as before. So add this to your code:

```
string Chosen_File = "";
```

```
openFD.InitialDirectory = "C:";  
openFD.Title = "Open a Text File";  
openFD.FileName = "";
```

The only thing we've changed here is the Title property. For the next line, we can add the Filters:

```
openFD.Filter = "Text Files|*.txt|Word Documents|*.doc";
```

The RichTextBox can open plain text files as well as Word documents, so we've added both of these to the Filter property. (It can't handle Word documents very well, though.)

The next thing to do is to display the Open File Dialogue box, so that a file can be selected. Add the following to your code:

```
if (openFD.ShowDialog() != DialogResult.Cancel)  
{  
    Chosen_File = openFD.FileName;  
    richTextBox1.LoadFile(Chosen_File, RichTextBoxStreamType.PlainText);  
}
```

This is more or less the same as before. But notice the line that adds the text file to RichTextBox:

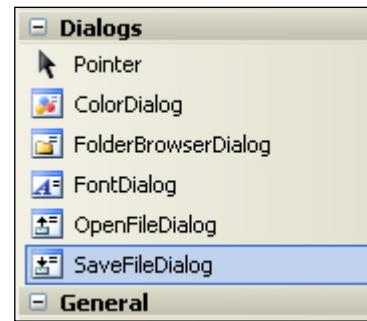
```
richTextBox1.LoadFile(Chosen\_File, RichTextBoxStreamType.PlainText);
```

You'll see a better way to open up a text file later in the course. For now, run your programme and test that it works. You should be able to add plain text file to the RichTextBox.

The Save menu

Another useful Method you can use with the RichTextBox is **SaveFile()**. As its name suggests, this allows you to save the file that's currently in the text box. We'll use this with another Dialog object. This time, we'll use the **SaveFileDialog** control instead of the OpenFileDialog control.

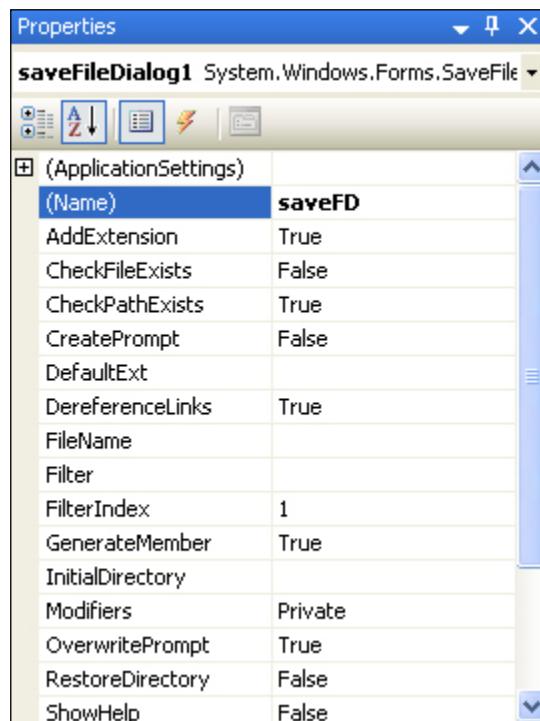
Return to your form, and locate the SaveFileDialog control in the Toolbox:



Double click to add one to your project. It should appear at the bottom of your screen:



Click on **saveFileDialog1** to select it. Now have a look at the Properties on the right hand side of the screen. Change the **Name** property to **saveFD**:



Now go back to your File menu, on your Menu Strip. Click on **File**, then double click your **Save** menu item. This will open up the code for this item:

```
private void mnuSave_Click(object sender, EventArgs e)
{
}
```

The code to add is practically the same as for the Open menu item. Instead of saying openFD, though, it's saveFD. Here it is:

```
private void mnuSave_Click(object sender, EventArgs e)
{
    string Saved_File = "";

    saveFD.InitialDirectory = "C:";
    saveFD.Title = "Save a Text File";
    saveFD.FileName = "";

    saveFD.Filter = "Text Files (*.txt)|All Files (*.*)";

    if (saveFD.ShowDialog() != DialogResult.Cancel)
    {
        Saved_File = saveFD.FileName;
        richTextBox1.SaveFile(Saved_File, RichTextBoxStreamType.PlainText);
    }
}
```

You should be able to work out what's happening, in the code above. The line that does the saving is this one:

```
richTextBox1.SaveFile(Saved_File, RichTextBoxStreamType.PlainText);
```

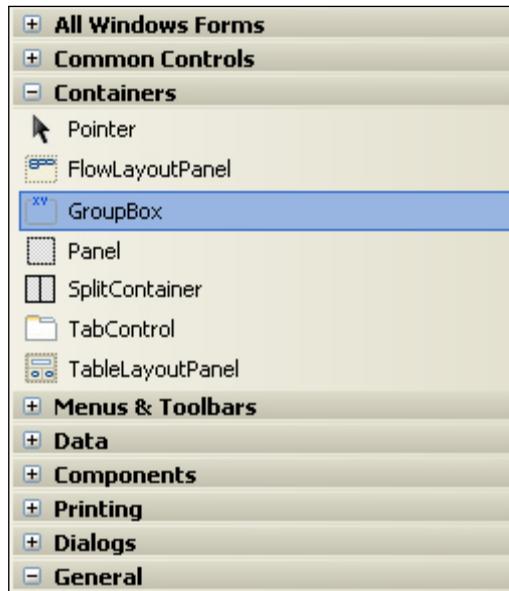
Again, though, there is a better way to manipulate files. You'll learn all about how to handle text files in a later section. For now, add the code above and Run your programme. Click your **File > Open** menu item to add a text file to your Rich Text Box. Make some changes to the text. Then click your **File > Save** menu item. You should find that the changes are permanent.

Checkboxes and Radio buttons

Checkboxes and Radio Buttons are way to offer your users choices. Checkboxes allow a user to select multiple options, whereas Radio Buttons allow only one. Let's see how to use them.

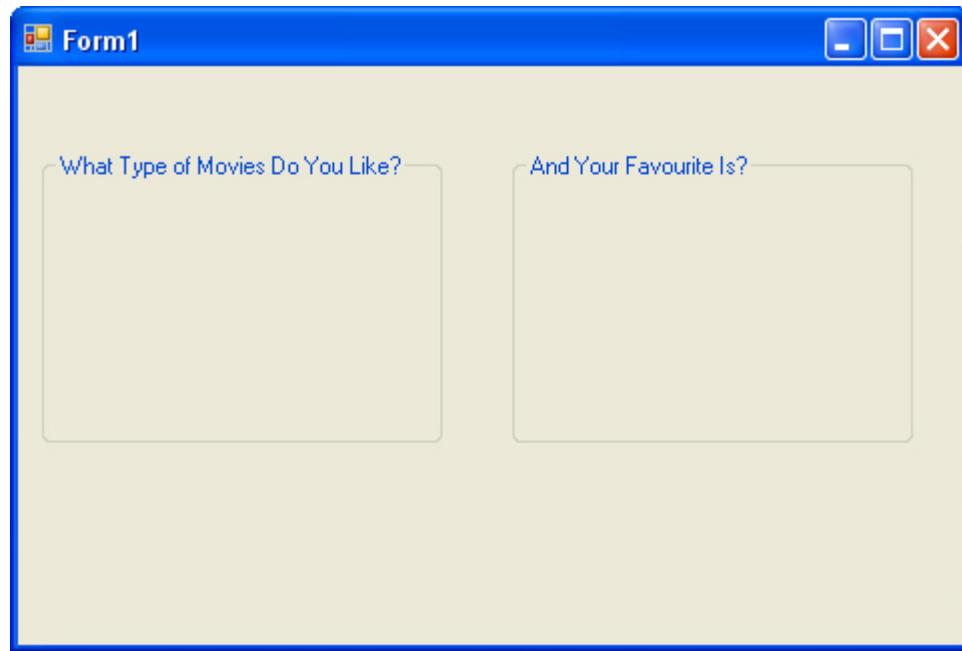
Start a new project. When your new form appears, make it nice and big. Because Checkboxes and Radio Buttons are small and fiddly to move around, its best to place them on a Groupbox. You can then move the Groupbox, and the checkboxes and radio buttons will move with them.

Locate the Groupbox control in the Toolbox on the left, under Containers. It looks like this:



Draw one out on your form. Locate the Text property in the properties window on the right of C#. Change the Text property to **What Type of Movies Do You Like?**

Add a second Groupbox along side of the first one, and set the Text property as **And Your Favourite Is?**. Your form will then look like this:



We'll place some Checkboxes on the first Groupbox, and some Radio Buttons on the second one.

Locate the **Checkbox** control on the toolbox, under **Common Controls**. Draw one out on your first Groupbox.

In the properties area on the right, notice that the default Name property is **checkBox1**. Leave it on this, but locate the **Text** property and change it to **Comedy**:



Draw four more checkboxes on the Groupbox, and set the Text properties as follows: **Action**, **Science Fiction**, **Romance**, **Animation**. (You can copy and paste the first one, instead of drawing them out.) Make the Text bold, and your Groupbox should look like this:



You add Radio Buttons in the same. So add five Radio Buttons to the second Groupbox. Leave the Name property on the defaults. But change the Text to the same as for the Checkboxes. Your form should look like ours below when you are finished:



Now add two buttons, one below each group box. Set the Text for the first one as **Selected Movies**. Set the Text for the second one as **Favourite Movie**. Here's what your form should look like now:



Run your form and test it out. What you should find is that you can select as many checkboxes as you like, but only one of the Radio Buttons.

Stop your programme and return to Design Time.

What we'll do now is to write code to get at which selections a user made. First, the Checkboxes.

Double click your **Selected Movies** button to open up the code window. Our code will make use of the **Checked** property of Checkboxes. This is either true or false. It will be true if the user places a check in the box, and false if there is no check.

We can use if statements to test the values of each checkbox. We only need to test for a true value:

```
if (checkBox1.Checked)
{
}
```

We can also build up a message, if an option was selected:

```
string movies = "";

if (checkBox1.Checked)
{
    movies = movies + checkBox1.Text;
}

MessageBox.Show(movies);
```

Inside of the if statement, we are building up the string variable we've called **movies**. We're placing the Text from the Checkbox into this variable.

Add a second if statement to your code:

```
string movies = "";

if (checkBox1.Checked)
{
    movies = movies + checkBox1.Text;
}

if (checkBox2.Checked)
{
    movies = movies + checkBox2.Text;
}

MessageBox.Show(movies);
```

The second if statement is the same as the first, except it refers to checkBox 2 instead of checkBox1.

Test out your code so far. Run your programme and check both boxes. Click your button and the message box should display the following:



As you can see, they are both on the same line, with no spacing.

Stop your programme and return to your code.

To get the choices on separate lines, there are a few ways you can do it. One way is to use the return and new line characters, like this:

```
movies = movies + checkBox1.Text + "\r\n";
```

The “\r” gets you a **Return** character, and the “\n” gets you a **Newline** character.

But you can also use the inbuilt Newline character. Like this:

```
movies = movies + checkBox1.Text + Environment.NewLine;
```

Newline is a property of the Environment class. As its name suggests, it adds a new line to your text.

Add one of the Newline options to both of your if statements, and then test it out. Your message box will look like this, with both options checked:



Return to your code, and add three more if statements. When you are finished, your coding window should look like this one:

```

private void button1_Click(object sender, EventArgs e)
{
    string movies = "";

    if (checkBox1.Checked)
    {
        movies = movies + checkBox1.Text + "\r\n";
    }

    if (checkBox2.Checked)
    {
        movies = movies + checkBox2.Text + "\r\n";
    }

    if (checkBox3.Checked)
    {
        movies = movies + checkBox3.Text + "\r\n";
    }

    if (checkBox4.Checked)
    {
        movies = movies + checkBox4.Text + "\r\n";
    }
    if (checkBox5.Checked)
    {
        movies = movies + checkBox5.Text + "\r\n";
    }

    MessageBox.Show(movies);
}

```

When you run your programme and check all the boxes, the message box will look like this, after the button is clicked:



To get at which Radio Button was chosen, the code is the same as for Checkboxes – just test the **Checked** state. The only difference is that you need **else if**, instead of 5 separate if statements:

```

string ChosenMovie = "";

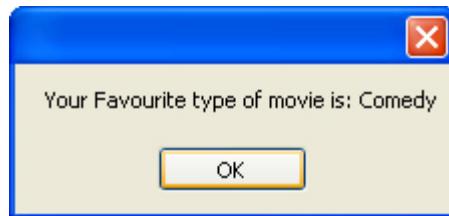
if (radioButton1.Checked)
{

```

```
    ChosenMovie = radioButton1.Text;  
}  
else if (radioButton2.Checked)  
{  
    ChosenMovie = radioButton2.Text;  
}
```

Exercise

Finish the code for your Radio Buttons by adding three more `else ... if` parts. Display a user's favourite movie type in a message box. When you've completed this exercise, your message box should look something like ours below:



OK, let's move on from Checkboxes and Radio buttons. In the next section, we'll take a look at how to Debug your code.

Error Handling and Debugging

Debugging refers to the process of trying to track down errors in your programmes. It can also refer to handling any potential errors that may occur. There are three types of errors that we'll take a look at:

- Design-time errors
- Run-time errors
- Logical errors

The longer your code gets, the harder it is to track down why things are not working. By the end of this section, you should have a good idea of where to start looking for problems. But bear in mind that debugging can be an art form in itself, and it gets easier with practice.

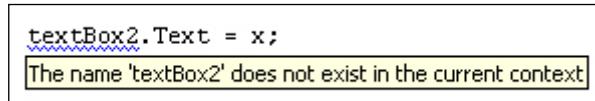
Errors at Design-Time

Design-Time errors are ones that you make before the programme even runs. In fact, for Design-Time errors, the programme won't run at all, most of the time. You'll get a popup message telling you that there were build errors, and asking would you like to continue.

Design-Time errors are easy enough to spot because the C# software will underline them with a wavy coloured line. You'll see three different coloured lines: blue, red and green. The blue wavy lines are known as **Edit and Continue** issues, meaning that you can make change to your code without having to stop the programme altogether. Red wavy lines are Syntax errors, such as a missing semicolon at the end of a line, or a missing curly bracket in an IF Statement. Green wavy lines are Compiler Warnings. You get these when C# spots something that could potentially cause a problem, such as declaring a variable that's never used.

Blue or Red Wavy Lines

In the image below, you can see that there's a blue wavy line under **textBox2** (later versions of Visual Studio may have red wavy lines, instead of blue ones):



This is an **Edit and Continue** Error. It's been flagged because the form doesn't have a control called **textBox2** – it's called **textBox1**. We can simply delete the 2 and replace it with a 1. The programme can then run successfully. Holding your

mouse over the wavy underline gives an explanation of the error. Some of these explanations are not terribly helpful, however!

Red Marks

These are Syntax errors. (Syntax is the “grammar” of a programming language, all those curly brackets and semicolons. Think of a Syntax error as the equivalent of programming spelling mistake.)

In the code below, we’ve missed out the semicolon at the end of the line:

```
textBox1.Text = x~~
```

Holding the mouse pointer over the red wavy line gives the following message:

```
textBox1.Text = x~~  
; expected
```

It’s telling us that a semicolon (;) is expected where the red wavy underline is.

In the next image, we’ve missed out a round bracket for the IF Statement:

```
if (x > 3~~  
{  
}  
) expected
```

Adding the round bracket will make the red wavy underline go away.

Green Wavy Lines

These are Compiler Warnings, the C# way of alerting you to potential problems. As an example, here’s some code that has a green wavy underline:

```
private void button1_Click(object sender, EventArgs e)  
{  
  
    string CompilerWarning = "";  
  
    textBox1.Text = "";  
  
}
```

Holding the mouse pointer over the green underlines gives the following message:

```
private void button1_Click(object sender, EventArgs e)
{
    string CompilerWarning = "";
    The variable 'CompilerWarning' is assigned but its value is never used
    textBox1.Text = "";
}
```

C# is flagging this because we have set aside some memory for the variable, but we're not doing anything with it.

This one is easy enough to solve, but some Compiler Errors can be a bit of a nuisance and the messages not nearly as helpful as the one above!

Whatever the colour of the underline, though, the point to bear in mind is this: C# thinks it has spotted an error in your code. It's up to you to correct it!

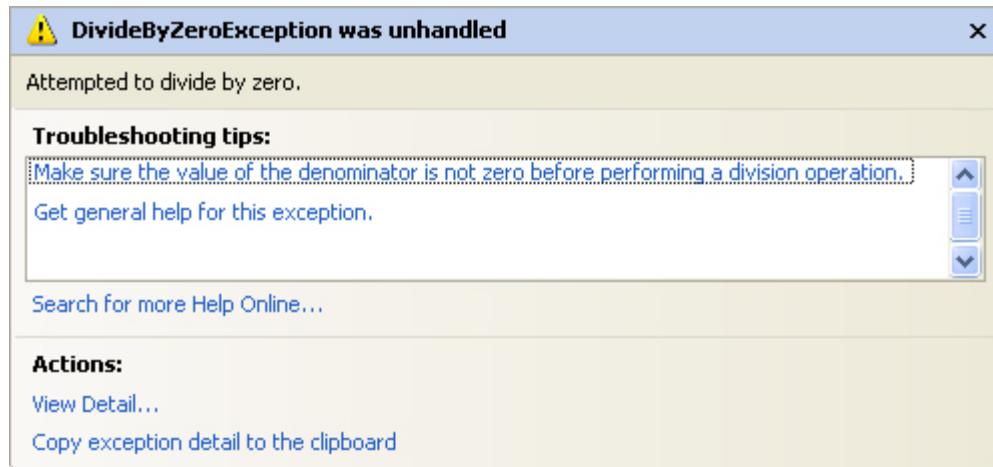
Run-Time Errors

Run-Time errors are ones that crash your programme. The programme itself generally starts up OK. It's when you try to do something that the error surfaces. A common Run-Time error is trying to divide by zero. In the code below, we're trying to do just that:

```
private void button1_Click(object sender, EventArgs e)
{
    int Num1 = 10;
    int Num2 = 0;
    int answer;

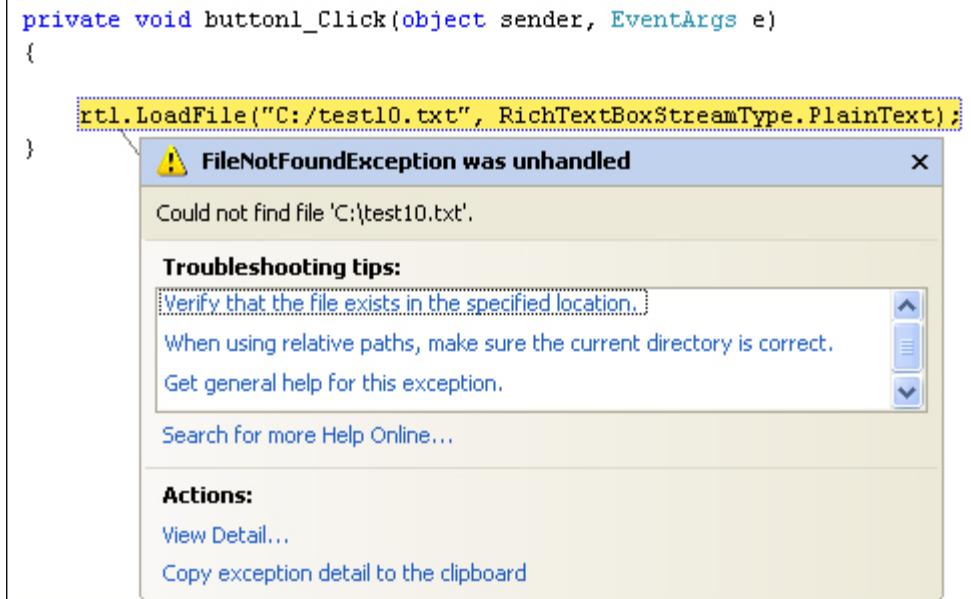
    answer = Num1 / Num2;
}
```

The programme itself reports no problems when it is started up, and there are no coloured wavy lines. When we click the button, however, we get the following error message (Visual Studio 2012 will have a plainer error message):



Had we left this in a real programme, it would just crash altogether (“bug out”). But if you see any error message like this one, it’s usually a Run-Time error.

Here’s another one. In the code below, we’re trying to open a file that doesn’t exist:



As the message explains, it can’t find the file called “C:/test10.txt”. Because we didn’t tell C# what to do if there was no such file, it just crashes.

Look out for this type of error message. It does take a bit of experience to work out what they mean; but some, like the one above, are quite straightforward.

You’ll see how to handle errors like this, soon. But there’s one final error type you have to know about.

Logic Errors

Logic errors are ones where you don't get the result you were expecting. You won't see any coloured wavy lines, and the programme generally won't "bug out" on you. In other words, you've made an error in your programming logic. As an example, take a look at the following code, which is attempting to add up the numbers one to ten:

```
private void button1_Click(object sender, EventArgs e)
{
    int startLoop = 11;
    int endLoop = 1;
    int answer = 0;

    for (int i = startLoop; i < endLoop; i++)
    {
        answer = answer + i;
    }

    MessageBox.Show("answer =" + answer.ToString());
}
```

When the code is run, however, it gives an answer of zero. The programme runs OK, and didn't produce any error message or wavy lines. It's just not the correct answer!

The problem is that we've made an error in our logic. The **startLoop** variable should be 1 and the **endLoop** variable 11. We've got it the other way round, in the code. So the loop never executes.

Logic errors can be very difficult to track down. To help you find where the problem is, C# has some very useful tools you can use. To demonstrate these tools, here's a new programming problem. We're trying to write a programme that counts how many times the letter "g" appears in the word "debugging".

Start a new C# Windows Application. Add a button and a textbox to your form. Double click the button, and then add the following code:

```

private void button1_Click(object sender, EventArgs e)
{
    int LetterCount = 0;
    string strText = "Debugging";
    string letter;

    for (int i = 0; i < strText.Length; i++)
    {
        letter = strText.Substring(i, 1);

        if (letter == "g")
        {
            LetterCount++;
        }
    }

    textBox1.Text = "g appears " + LetterCount + " times";
}

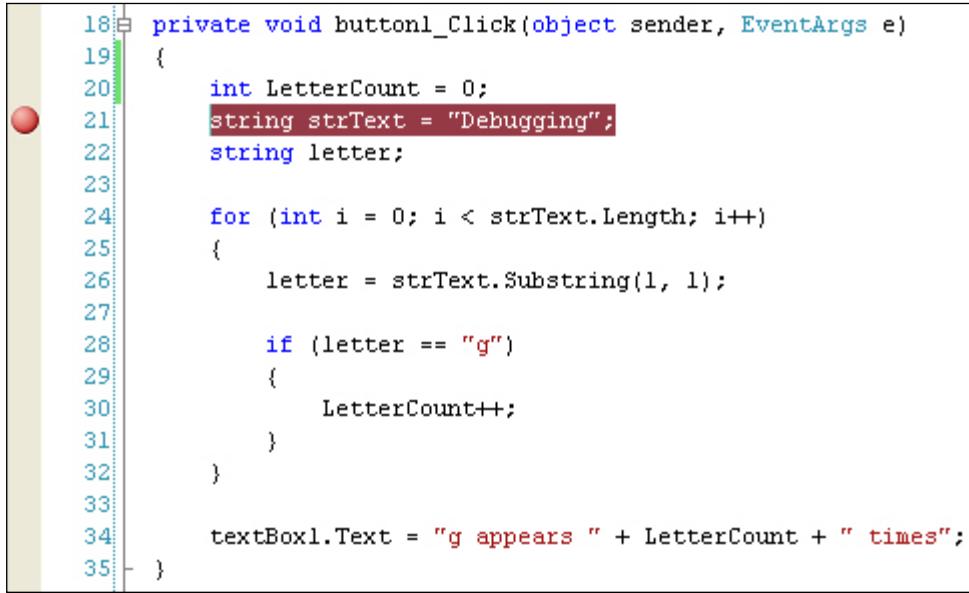
```

The answer should, of course, be 3. Our programme insists, however, that the answer is zero. It's telling us that there aren't any g's in Debugging. So we have made a logic error, but where?

Breakpoints

The first debugging tool we'll look at is the Breakpoint. This is where you tell C# to halt your code, so that you can examine what is in your variables. They are easy enough to add.

To add a Breakpoint, all you need to do is to click in the margins to the left of a line of code:



The screenshot shows a code editor window with the following C# code:

```

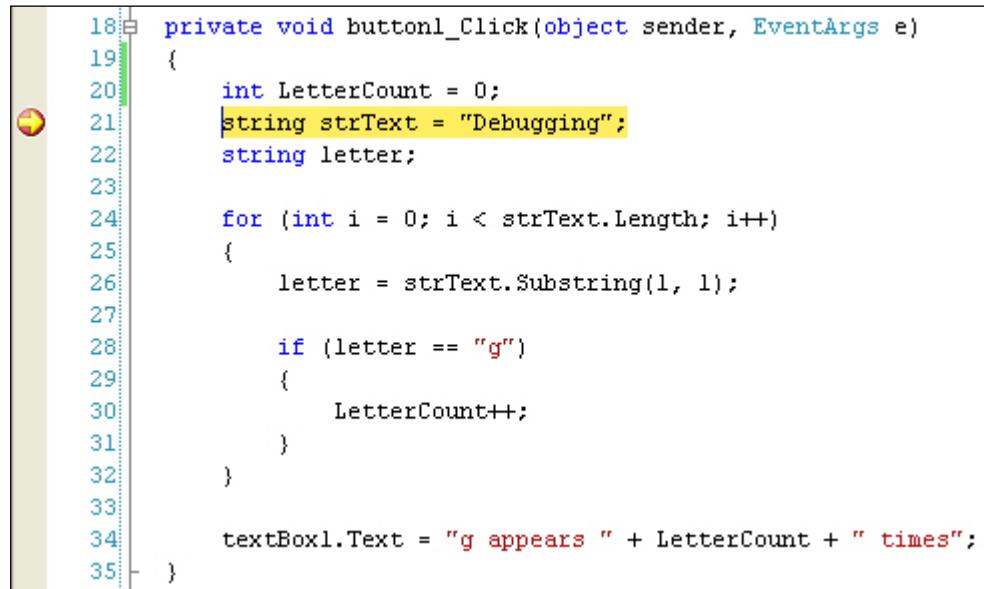
18: private void button1_Click(object sender, EventArgs e)
19: {
20:     int LetterCount = 0;
21:     string strText = "Debugging";
22:     string letter;
23:
24:     for (int i = 0; i < strText.Length; i++)
25:     {
26:         letter = strText.Substring(i, 1);
27:
28:         if (letter == "g")
29:         {
30:             LetterCount++;
31:         }
32:     }
33:
34:     textBox1.Text = "g appears " + LetterCount + " times";
35: }

```

A red circular breakpoint icon is positioned in the margin column to the left of the first line of code (line 18). The code editor interface includes a vertical yellow bar on the far left and a green vertical bar near the top of the code area.

In the image above, we clicked in the margins, just to the left of line 21. A reddish circle appears. Notice too that the code on the line itself gets highlighted.

To see what a breakpoint does, run your programme and then click your button. C# will display your code:



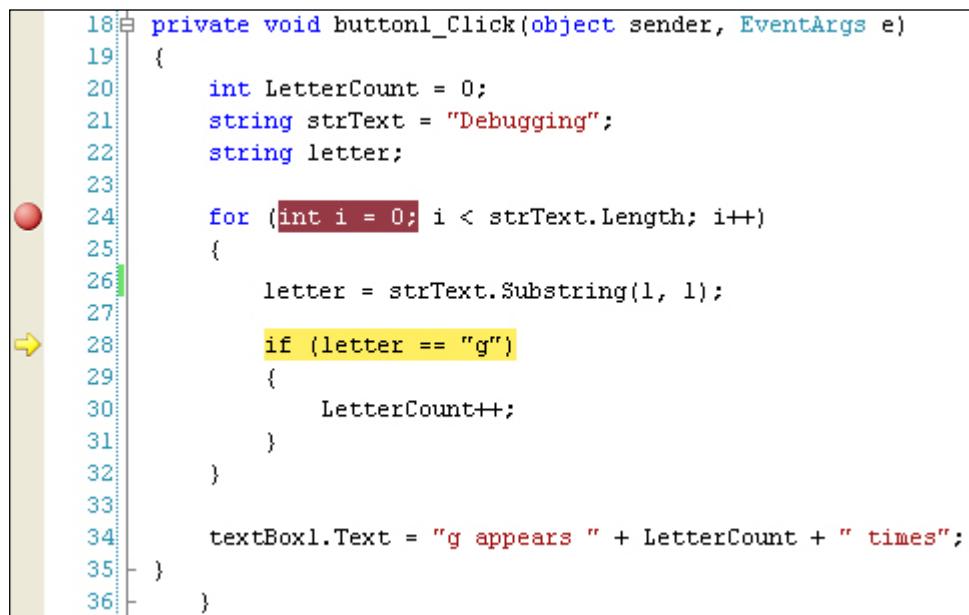
```

18: private void button1_Click(object sender, EventArgs e)
19: {
20:     int LetterCount = 0;
21:     string strText = "Debugging";
22:     string letter;
23:
24:     for (int i = 0; i < strText.Length; i++)
25:     {
26:         letter = strText.Substring(1, 1);
27:
28:         if (letter == "g")
29:         {
30:             LetterCount++;
31:         }
32:     }
33:
34:     textBox1.Text = "g appears " + LetterCount + " times";
35: }

```

There will be a yellow arrow on top of your red circle, and the line of code will now be highlighted in yellow. (If you want to enable line numbers in your own code, click **Tools > Options** from the C# menus at the top. On the Options box, click the plus symbol next to **Text Editor**, then **C#**. Click on **General**. On the right hand side, check the box for Line Numbers, under the **Display** heading.)

Press F10 on your keyboard and the yellow arrow will jump down one line. Keep pressing F10 until line 28 in your code is highlighted in yellow, as in the image below:



```

18: private void button1_Click(object sender, EventArgs e)
19: {
20:     int LetterCount = 0;
21:     string strText = "Debugging";
22:     string letter;
23:
24:     for (int i = 0; i < strText.Length; i++)
25:     {
26:         letter = strText.Substring(1, 1);
27:
28:         if (letter == "g")
29:         {
30:             LetterCount++;
31:         }
32:     }
33:
34:     textBox1.Text = "g appears " + LetterCount + " times";
35: }

```

Move your mouse pointer over the **letter** variable and C# will show you what is currently in this variable:

```
for (int i = 0; i < strText.Length; i++)
{
    letter = strText.Substring(1, 1);
    if (letter == "e")
    {
        LetterCount++;
    }
}
```

Now hold your mouse over **strText** to see what is in this variable:

```
for (int i = 0; i < strText.Length; i++)
{
    letter = strText.Substring(1, 1);
    if (letter == "D")
    {
        LetterCount++;
    }
}
```

Although we haven't yet mentioned anything about the Substring method, what it does is to grab characters from text. The first 1 in between the round brackets means start at letter 1 in the text; the second 1 means grab 1 character. Starting at letter 1, and grabbing 1 character from the word Debugging, will get you the letter "D". At least, that's what we hoped would happen!

Unfortunately, it's grabbing the letter "e", and not the letter "D". The problem is that the Substring method starts counting from zero, and not 1.

Halt your programme and return to the code. Change your Substring line to this:

letter = strText.Substring(0, 1);

So type a zero as the first number of Substring instead of a 1. Now run your code again:

```
for (int i = 0; i < strText.Length; i++)
{
    letter = strText.Substring(0, 1);
    if (letter == "D")
    {
        LetterCount++;
    }
}
```

This time, the correct letter is in the variable. Halt your programme again. Click your Breakpoint and it will disappear. Run the programme once more and it will run as it should, without breaking.

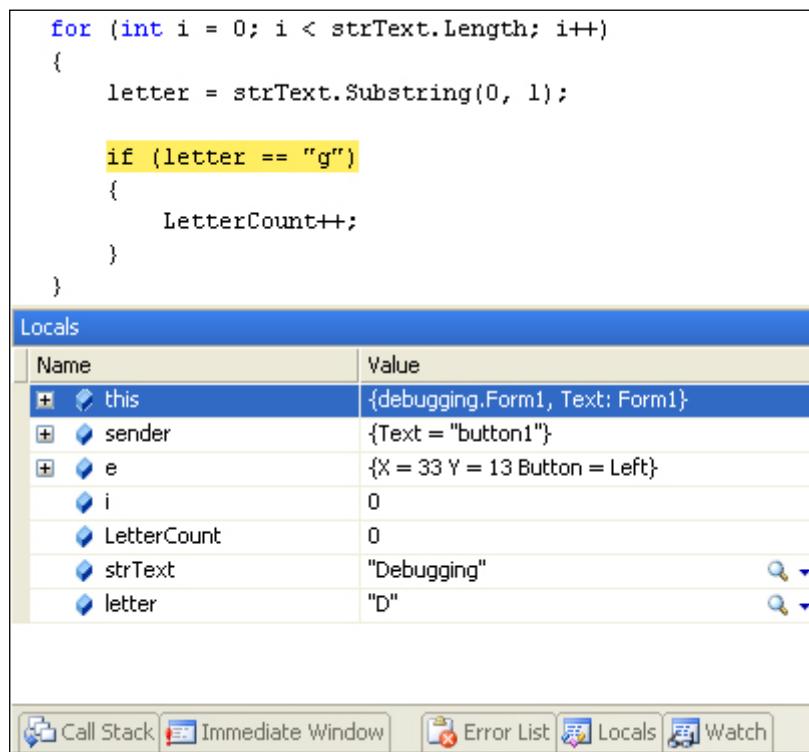
So have we solved the problem? Is the programme counting the letter g's correctly?

No! The letter count is still zero! So where's the error? To help you track it down, there's another tool you can use – the Locals Window.

The Locals Window

The Locals Window keeps track of what is in local variables (variables you've set up in this chunk of code, and not outside it).

Add a new breakpoint, this time in the margins to the left of your IF statement. Run your programme again, and click the button. When you see the yellow highlighted line, click the **Debug** menu at the top of C#. From the Debug menu, click **Windows > Locals**. You should see the following window appear at the bottom of your screen:

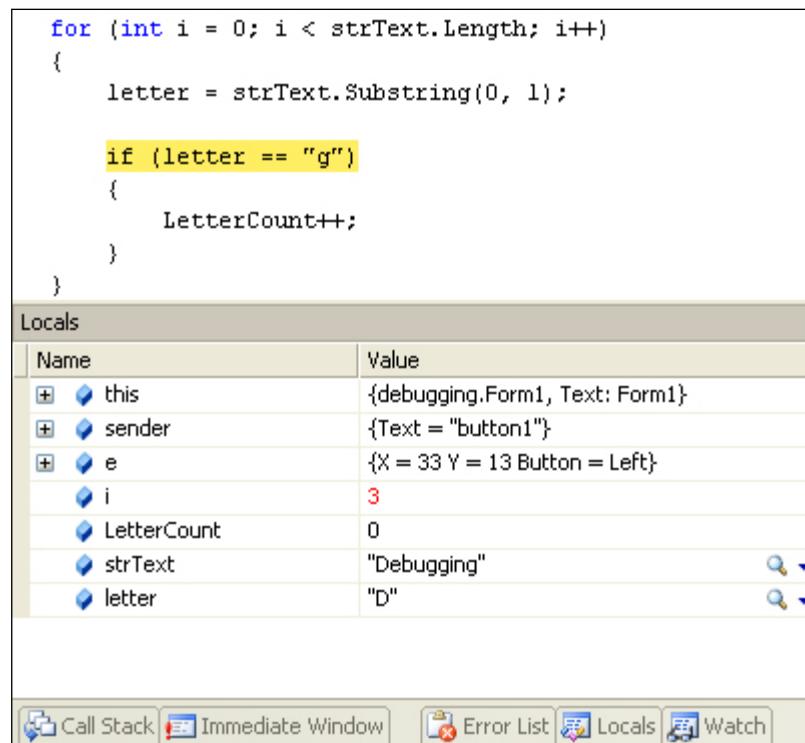


The screenshot shows a code editor with a portion of C# code and a 'Locals' window. The code highlights the condition 'letter == "g"'. The 'Locals' window lists variables with their current values:

| Name | Value |
|-------------|--------------------------------|
| this | {debugging.Form1, Text: Form1} |
| sender | {Text = "button1"} |
| e | {X = 33 Y = 13 Button = Left} |
| i | 0 |
| LetterCount | 0 |
| strText | "Debugging" |
| letter | "D" |

At the bottom of the Locals window are tabs for Call Stack, Immediate Window, Error List, Locals (which is selected), and Watch.

Keep pressing F10 and the values will change. Here's what is inside of the variables after a few spins round the loop:



The variable **i** is now 3; **letter** is still “D”, and **LetterCount** is still 0. Keep pressing F10 and go round the loop a few times. What do you notice? Keep your eye on what changes in your Locals window. The changes should turn red.

You should notice that the value in **i** changes but **letter** never moves on. It is “D” all the time. And that’s why LetterCount never gets beyond 0. But why does it never move on?

Exercise I

Why does LetterCount never gets beyond 0? Correct the code so that your textbox displays the correct answer of 3 when the programme is run. HINT: think Substring and loops!

[Answer to Exercise I](#)

Try ... Catch

C# has some inbuilt objects you can use to deal with any potential errors in your code. You tell C# to **Try** some code, and if can’t do anything with it you can **Catch** the errors. Here’s the syntax:

```

try
{
}
catch
{
}

```

In the code below, we're trying to load a text file into a RichTextBox called **rtb**:

```
try
{
    rtb.LoadFile("C:/test.txt");
}
catch (System.Exception excep)
{
    MessageBox.Show(excep.Message);
}
```

The code we want to execute goes between the curly brackets of **try**. We know that files go missing, however, and want to trap this “File not Found” error. We can do that in the **catch** part. Note what goes between the round brackets after **catch**:

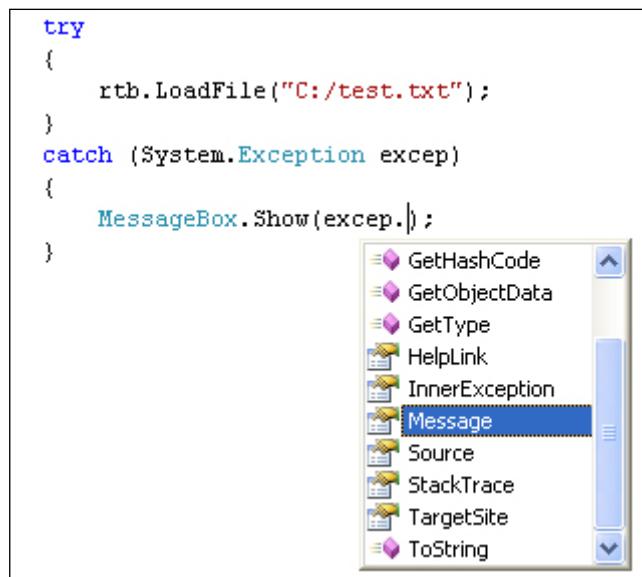
System.Exception excep

Exception is the inbuilt object that handles errors, and this follows the word **System** (known as a namespace). After System.Exception, you type a space. After the space, you need the name of a variable (**excep** is just a variable name we made up and, like all variable names, you call it just about anything you like).

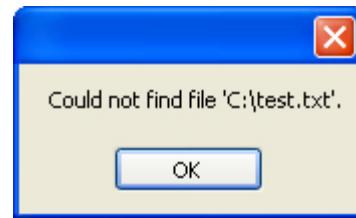
The code between the curly brackets of **catch** is this:

MessageBox.Show(**excep.Message**);

So we’re using a message box to display the error. After the variable name (**excep** for us), type a dot and you’ll see the IntelliSense list appear:



If you just want to display the inbuilt system message, select **Message** from the list. When the programme is run, you’ll see a message box like this:



If you know the type of error that will be generated, you can use that instead:

```
catch (System.IO.FileNotFoundException)
{
    MessageBox.Show("File not found");
}
```

To find out what type of error will be generated, use this:

```
catch (System.Exception excep)
{
    MessageBox.Show(excep.GetType().ToString());
}
```

The message box will tell you what system error is being generated. You can then use this between the round brackets of **catch**.

If you want to keep things really simple, though, you can miss out the round brackets after **catch**. In the code below, we're just creating our own error messages:

```
try
{
    rtb.LoadFile("C:/test.txt");
}
catch
{
    MessageBox.Show("An error occurred");
}
```

You can add more catch parts, if you want:

```
try
{
    rtb.LoadFile("C:/test.txt");
}
catch
{
    MessageBox.Show("An error occurred");
}
catch
{
    MessageBox.Show("Couldn't find the file");
```

```
        }
    catch
    {
        MessageBox.Show("Or maybe it was something else!");
    }
```

The reason you would do so, however, is if you think more than one error may be possible. What if the file could be found but it can't be loaded into a RichTextBox? In which case, have two **catch** blocks, one for each possibility.

There's also a **Finally** part you can add on the end:

```
try
{
    rtb.LoadFile("C:/test.txt");
}
catch (System.Exception excep)
{
    MessageBox.Show(excep.Message);
}
finally
{
    //CLEAN UP CODE HERE
}
```

You use a Finally block to clean up. (For example, you've opened up a file that needs to be closed.) A **Finally** block will always get executed, whereas only one **catch** will.

(NOTE: there is also a **throw** part to the catch blocks for when you want a more specific error, want to throw the error back to C#, or you just want to raise an error without using **try ... catch**. Try not to worry about **throw**.)

We won't be using **Try ... Catch** block too much throughout this book, however, because they tend to get in the way of the explanations. But you should try and use them in your code as much as possible, especially if you suspect a particular error may crash your programme.

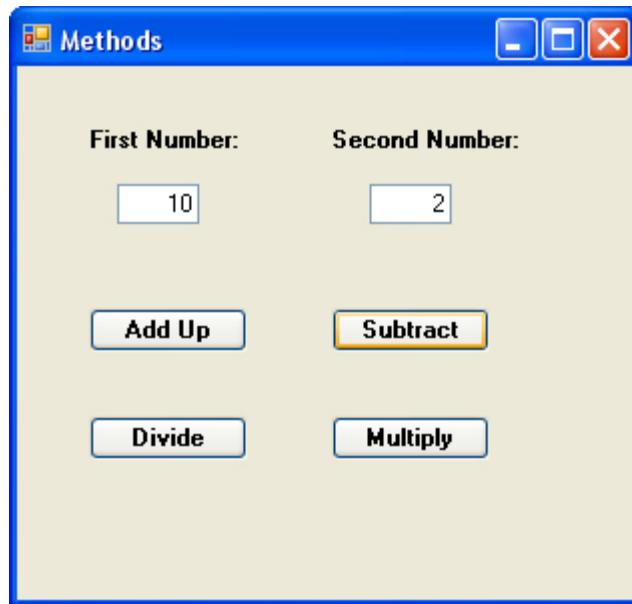
Methods

So far, all of your programming code has gone between the curly brackets of buttons. But this is not an effective way to programme. If you keep all your code in one place, it will become more and more unreadable the longer it gets. Instead, you can use something called a Method.

A Method is just a segment of code that does a particular job. Think about the calculator programme you have been working on. You can have one Method (a chunk of code) to add up, one to subtract, another one to divide, and a fourth Method to multiply. The idea is that when you want to add up, you just call the **Add Up** Method into action.

To get you started with Methods, we'll create a simple programme that takes two numbers from text boxes. We'll have four buttons, one to add up, one to subtract, one to divide, and one to multiply. We'll use Methods to do the calculating. Off we go then!

Create a new C# project, and design the following form:



You can keep the buttons and text boxes on their default names (button1, button2, textbox1, textbox2, etc.)

Double click the **Add Up** button to open up the coding window. The cursor will be flashing inside of the button code. However, you create Methods outside of any other code. So move the cursor after the final curly bracket of the button code. Then hit your enter key a few times to give yourself some space. Type the following:

```
void AddUp()
{
    MessageBox.Show("Add Up Here");

    return;
}
```

Your coding window will then look like ours below:

```
private void button1_Click(object sender, EventArgs e)
{
}

void AddUp()
{
    MessageBox.Show("Add Up Here");

    return;
}
```

Methods can [return](#) a value, such as the answer to the addition. But they don't have to. Our Method above just displays a message box when it is called into action. If you don't want anything back from your Methods, you set them up by typing the keyword [void](#). After a space, you need to come up with a name for your Method. We've called ours [AddUp](#). But they are just the same as variable names, and you call them almost anything you like. (The same rules apply to naming Methods as they do to naming Variables.)

After coming up with a name for your Method, you type a pair of round brackets. You can put things between the round brackets, and you'll see how to do that shortly.

After the round brackets, you need a pair of curly brackets. The code for your Method goes between the curly brackets. For us, this was just a Message Box.

Before the final curly bracket, we've typed the word [return](#), followed by a semicolon. This is not necessary, if you've set your Method up as [void](#), since you don't want it to return anything: you just want it to get on with its job. We've added the [return](#) keyword because it's just standard practice. But when C# sees the [return](#) keyword, it will break out of your Method. If you type any code after that, it won't get executed. (You'll see a different way to use the [return](#) keyword when we want to get something back from a Method.)

Calling your Methods

Our Method is not doing much good at the moment, since it's not being called into action anywhere. We'll get it to do its work when a button is clicked.

To call a Method, you just do this:

AddUp();

So you type the name of your Method, along with the round brackets. The semicolon ends the line of code, as normal.

So add that line to your button that Adds Up:

```
private void button1_Click(object sender, EventArgs e)
{
    AddUp();
}

void AddUp()
{
    MessageBox.Show("Add Up Here");

    return;
}
```

Run your programme and test it out. Click the Add Up button and you should see the message box display.

What happens is that the button calls the **AddUp** Method into action. C# then trots off and executes all of the code for your Method. It then comes back to the line where it was called, ready to execute any other code you may have for your button.

Passing values to your Methods

You can hand values over to your Methods. The Method can then use these values in its code. For us, we want to get two numbers from the text boxes, and then add them up. The two values from the text boxes, then, need to be handed over to our Method. The code to add up will go between the curly brackets of the AddUp Method.

To hand values over to your Methods, you place them between the round brackets. These are called parameters. (You'll also hear the term **arguments**, often used to mean the same thing. There is a subtle difference, however, which you'll see shortly. It's not crucial that you learn the difference, though!)

Change your Method to this:

```
void AddUp(int firstNumber, int secondNumber)
{
    MessageBox.Show("Add Up Here");

    return;
}
```

So we've added two parameters between the round brackets of AddUp. A parameter is set up just like an ordinary variable. You start with the variable type (int, string, bool, etc.) then a space. After the space, you need to come up with a name for your parameter. We've called our first parameter **firstNumber**. But we could have called it almost anything. If you want more than one parameter, you separate them with commas. We've added a second parameter and called it **secondNumber**. Both parameters have been set up as type **int**. They're going to hold numbers, in other words.

We can use these parameters in the code for the Method. Adapt your **AddUp** code so that it's like ours below:

```
void AddUp(int firstNumber, int secondNumber)
{
    int answer;

    answer = firstNumber + secondNumber;

    MessageBox.Show(answer.ToString());

    return;
}
```

So we've set up a new **int** variable called **answer**. We're then adding up the variables **firstNumber** and **secondNumber**. The result goes in the new variable. The message box displays what is in the variable called **answer**.

If you try to run your code now, however, you'll get an error. There will be a wavy blue line under **AddUp**, along with a strange error message:

```
private void button1_Click(object sender, EventArgs e)
{
    AddUp();
}
```

No overload for method 'AddUp' takes '0' arguments

This error message can be translated as "You have no Method called AddUp that takes zero arguments." When you're calling a Method into action, you need to use the same number of parameters (now called arguments instead) as when you set it up. We set up our Method to take two values, **firstNumber** and **secondNumber**. So we need to use two values when we call the Method.

Here's the difference between an argument and a parameter: It's a parameter when you set up the values in the method; It's an argument when you're calling it (passing the values to the Method).

Change your button code to this:

```
private void button1_Click(object sender, EventArgs e)
{
    AddUp(15, 5);
}
```

So we've now typed two numbers between the round brackets, 15 and 5. The first value you type will get handed to parameter one of your Method, the second value will get handed to parameter two, and so on. The picture below might clear things up, if all of that is a little confusing:

```
private void button1_Click(object sender, EventArgs e)
{
    AddUp(15, 5);
}

void AddUp(int firstNumber, int secondNumber)
{
}
```

So the Method itself has two parameters. When it is being called in the button code there are now two arguments, once for each parameter.

Run your programme again and there shouldn't be any errors. When you click your button, you should see the answer to the addition.

Halt your programme, and change your button code to this:

```
private void button1_Click(object sender, EventArgs e)
{
    AddUp( int.Parse(textBox1.Text), int.Parse(textBox2.Text) );
}
```

Now, the values between the round brackets are no longer numbers that we've just typed in there. Instead, we're getting the values from the text boxes, and placing them between the round brackets. Note the comma separating the two values.

You can also do this:

```
private void button1_Click(object sender, EventArgs e)
{
    int number1;
    int number2;

    number1 = int.Parse(textBox1.Text);
    number2 = int.Parse(textBox2.Text);

    AddUp(number1, number2);
}
```

We're now putting the values from the text boxes into two new variables, called **number1** and **number2**. When we call the Method, we can use these variable names:

```
AddUp( number1, number2 );
```

The values in these two variables will get handed to our Method. But all you are trying to do is to pass two integers over to your Method. You need two integers because that's the way you set the Method up.

One more thing to note, here. When we set the Method up, the two parameters were called **firstNumber** and **secondNumber**. When we called it from the button, however, the two variables are called **number1** and **number2**. So we've used different variables names. This is perfectly OK, and C# doesn't get confused. All that matters is that you are passing the correct information over to the Method.

Getting values back from Methods

The Method we set up used the keyword **void**. We used void because we didn't want anything back from the Method. But quite often you will want something back from your Methods.

What we'll do now is to use the Subtract button and deduct one text box number from the other. We'll set up another Method called Subtract. This time, we'll set it up so as to return an answer.

If you want to return a value from your Methods, you can't use the keyword **void**. Simply because void means "Don't return an answer". Instead of using void, we'll use the keyword **int**.

Add the following Method to your code, either above or below the AddUp Method:

```
private int Subtract(int firstNumber, int secondNumber)
{
    int answer;

    answer = firstNumber - secondNumber;

    return answer;
}
```

If you add a comment, your coding window should look like ours:

```

private void button1_Click(object sender, EventArgs e)
{
    int number1;
    int number2;

    number1 = int.Parse(textBox1.Text);
    number2 = int.Parse(textBox2.Text);

    AddUp(number1, number2);
}

//=====
//          METHODS
//=====

private int Subtract(int firstNumber, int secondNumber)
{
    int answer;

    answer = firstNumber - secondNumber;

    return answer;
}

void AddUp(int firstNumber, int secondNumber)
{
    int answer;

    answer = firstNumber + secondNumber;

    MessageBox.Show(answer.ToString());

    return;
}
}

```

So we have one button and two Methods. Before we explain the new Method, double click the Subtract button on your form to get at its code. Then add the following:

```

private void button2_Click(object sender, EventArgs e)
{
    int number1;
    int number2;
    int returnValue = 0;

    number1 = int.Parse(textBox1.Text);
    number2 = int.Parse(textBox2.Text);

    returnValue = Subtract(number1, number2);

    MessageBox.Show(returnValue.ToString());
}

```

We'll explain how this button code works in a moment. But run your programme and you should see a message box appear when you click your Subtract button. Hopefully it will have the right answer!

Now have a look at the first line of the new Method:

```
private int Subtract(int firstNumber, int secondNumber)
```

The part in round brackets is exactly the same as before, and works the same way – set up the Method to accept two integer values. What's new is this part:

private int Subtract

Subtract is just the name of the Method, something we came up with ourselves. Before the Method name, however, we have two new keywords – **private** and **int**.

What we want our Method to do is to bring back the answer to our subtraction. The answer will obviously be a number. And that's why **int** comes before the Method name: we want the answer to the Subtract Method to be an **integer**. If you want to return values from your Methods they need what's called a **return type**. This is a variable like **int**, **float**, **string**, **bool**, etc. What you're telling C# to do is to **return** an **int** (or a **bool**, or a **float**).

Have a look at the whole Method again:

```
private int Subtract(int firstNumber, int secondNumber)
{
    int answer;

    answer = firstNumber - secondNumber;

    return answer;
}
```

Notice the final line:

```
return answer;
```

This means, “**return** whatever is inside of the variable called **answer**.”

But where is C# returning to? Here's the code for the Subtract button again. The important line is in red bold below:

```
private void button2_Click(object sender, EventArgs e)
{
    int number1;
    int number2;
    int returnValue = 0;

    number1 = int.Parse(textBox1.Text);
```

```

        number2 = int.Parse(textBox2.Text);

returnValue = Subtract(number1, number2);

        MessageBox.Show(returnValue.ToString());
    }

```

When you click the button on the form, C# moves down line by line. When it gets to this line:

```
returnValue = Subtract(number1, number2);
```

it will trot off and locate the Method called Subtract. It will then try to work out the code for the Method. Once it has an answer, it comes back to the same place. We have the call to the Method after an equals sign. Before the equals sign we have a new integer variable, which we've called **returnValue**. C# will store the answer to the Subtract Method inside of this **returnValue** variable. In other words, it's just like a normal variable assignment: work out the answer on the right of the equals sign, and store it on the left. In case that's not clear, these diagrams may help:

```

private void button2_Click(object sender, EventArgs e)
{
    int number1;
    int number2;
    int returnValue = 0;

    number1 = int.Parse(textBox1.Text);
    number2 = int.Parse(textBox2.Text);

    returnValue = Subtract(number1, number2);
    MessageBox.Show(returnValue.ToString());
}

```

Start here

```

private int Subtract(int firstNumber, int secondNumber)
{
    int answer;

    answer = firstNumber - secondNumber;

    return answer;
}

```

Jump to here

```

private int Subtract(int firstNumber, int secondNumber)
{
    int answer;

    answer = firstNumber - secondNumber;

    return answer;
}

```

Store the answer to the Method here

```

private void button2_Click(object sender, EventArgs e)
{
    int number1;
    int number2;
    int returnValue = 0;

    number1 = int.Parse(textBox1.Text);
    number2 = int.Parse(textBox2.Text);

    returnValue = Subtract(number1, number2);

    MessageBox.Show(returnValue.ToString());
}

```

Jump back to here with the answer

```

private void button2_Click(object sender, EventArgs e)
{
    int number1;
    int number2;
    int returnValue = 0;

    number1 = int.Parse(textBox1.Text);
    number2 = int.Parse(textBox2.Text);

    returnValue = Subtract(number1, number2);

    MessageBox.Show(returnValue.ToString());
}

```

Store the return value here

After those steps, C# then drops down to the next line, which for us is a message box.

It can be tricky trying to follow what the method is doing, and what gets passed back. But just remember these points:

- To set up a Method that returns a value, use a return type like `int`, `float`, `bool`, `string`, etc
- Use the keyword `return`, followed by the answer you want to have passed back
- Store the answer to your Method in another variable, which should come before an equals sign

One thing we haven't explained is why we started our Method with the word `private`.

Private refers to which other code has access to the Method. By using the `private` keyword you're telling C# that the Method can't be seen outside of this particular class. The class in question is the one at the top of the code, for the form. This one:

```
public partial class Form1 : Form
```

An alternative to `private` is `public`, which means it can be seen outside of a particular class or method. (There's also a keyword called `static`, which we'll cover later in the course.)

We'll leave Methods for now. But we'll be using them a lot for the rest of this book! To help your understanding of the topic, try this exercise.

Exercise J

Add two more Methods to your code, one to Multiply, and one to Divide. Add code to your Multiply and Divide buttons that uses your new Methods. When you run your programme, all four buttons should work.

[Answer to Exercise J](#)

Understanding Arrays

Arrays are another of those concepts that you need to get the hang of, if you're to programme effectively. In this section, you'll see you what arrays are, and how to use them.

What is an Array?

The variables you have been working with so far have only been able to hold one value at a time. Your integer variables can only hold one number, and your strings one chunk of text. An array is a way to hold more than one variable at a

time. Think of a lottery programme. If you're just using single variables, you'd have to set up your lottery numbers like this:

```
lottery_number_1 = 1;
lottery_number_2 = 2;
lottery_number_3 = 3;
lottery_number_4 = 4;
etc
```

Instead of doing that, an array allows you to use just one identifying name that refers to lots of numbers.

How to set up an Array

You set up an array like this:

```
int[] lottery_numbers;
```

So you start with the type of array you need. In the line above, we're telling C# that the array will hold numbers (**int**). After the array type, you need a pair of square brackets. There should be no space between the array type and the brackets. After the square brackets, type a space and then the name you want to use for your array, **lottery_numbers** in our case.

If your array needs to hold floating point numbers, you'd set your array up like this:

```
float[] my_float_values;
```

An array that needs to hold text would be set up like this:

```
string[] my_strings
```

So it's pretty much just like setting up a normal variable, except you type a pair of square brackets after **int**, or **float**, or **string**.

The next thing you need to do is to tell C# how big your array will be. The size of an array is how many items it is going to hold. You do it like this:

```
lottery_numbers = new int[49];
```

So the name of your array goes before an equals sign (=). After the equals sign, you type the word **new**. This tells C# that it is a new object. After a space, you need the array type again (**int** for us). Next comes some square brackets. This time, however, you type the size of the array between the brackets. In the code above, we're saying that the array will hold 49 numbers.

So the two lines would be:

```
int[] lottery_numbers;
lottery_numbers = new int[49];
```

If you prefer, you can put all that on one line:

```
int[] lottery_numbers = new int[49];
```

But you're doing two things at once, here: before the equals sign, you're telling C# that you want to set up an array; after the equals sign, you're creating a new array object of a particular size.

Assigning values to your arrays

So far, you have just set up the array, and created an array object. But the array doesn't yet hold any values. (Well it does, because C# assigns some default values for you. In the case of `int` arrays, this will be just zeros. But they're not your values!)

To assign a value to an array, you use the square brackets again. Here's the syntax:

```
array_name[position_in_array] = array_value;
```

So you start with the name of your array, followed by a pair of square brackets. In between the square brackets, you need a position in your array. You then type an equals sign, and the value that is going in that position. Here's an example using our lottery numbers:

```
lottery_numbers[0] = 1;
lottery_numbers[1] = 2;
lottery_numbers[2] = 3;
lottery_numbers[3] = 4;
```

First of all, note that the first position in a C# array is zero, and not 1. This is slightly confusing, and can trip you up! But we're telling C# to assign a value of 1 to the first position in the array, a value of 2 in the second position, a value of 3 in the third, and so on. Just bear in mind that array positions start at 0.

Another way to assign values in array is by using curly brackets. If you only have a few values going in to the array, you could set it up like this:

```
int[] lottery_numbers = new int[4] { 1, 2, 3, 4 };
```

So we've set up the array the same way as before – all on one line. This time, we have a pair of curly brackets at the end. In between the curly brackets, type the values for your array, and separate each value with a comma.

Arrays and Loops

Arrays come into their own with loops. The idea is that you can loop round each position in your array and access the values. We'll try a programming example, this time.

So start your C# software up, if you haven't already, and create a new Windows Application. Add a button and a list box to your form. Double click your button to get at the code. For the first line, add code to clear the list box:

```
private void button1_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();
}
```

For the second and third lines, set up an integer array:

```
private void button1_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();

    int[] lottery_numbers;
    lottery_numbers = new int[4];
}
```

Now add some values to each position in the array:

```
private void button1_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();

    int[] lottery_numbers;
    lottery_numbers = new int[4];

    lottery_numbers[0] = 1;
    lottery_numbers[1] = 2;
    lottery_numbers[2] = 3;
    lottery_numbers[3] = 4;
}
```

If you wanted to, you could display each number in the list box like this:

```
listBox1.Items.Add( lottery_numbers[0] );
listBox1.Items.Add( lottery_numbers[1] );
listBox1.Items.Add( lottery_numbers[2] );
```

```
listBox1.Items.Add(lottery_numbers[3]);
```

So to get at the value in an array, you just use the array name and a position number, known as the index number:

```
lottery_numbers[0];
```

This is enough to display what value is at this position in the array. Try it out. Add the list box code to your programme:

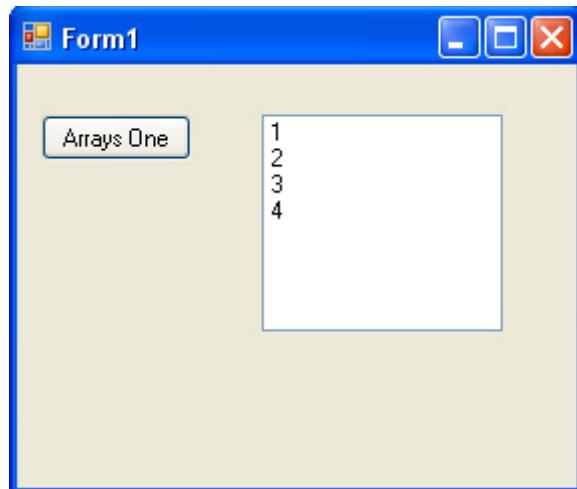
```
private void button1_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();

    int[] lottery_numbers;
    lottery_numbers = new int[4];

    lottery_numbers[0] = 1;
    lottery_numbers[1] = 2;
    lottery_numbers[2] = 3;
    lottery_numbers[3] = 4;

    listBox1.Items.Add(lottery_numbers[0]);
    listBox1.Items.Add(lottery_numbers[1]);
    listBox1.Items.Add(lottery_numbers[2]);
    listBox1.Items.Add(lottery_numbers[3]);
}
```

Run your programme and click your button. Your form should look like this:



So the numbers 1 to 4, the values we placed in the array, are now displayed in the list box. Halt your programme and return to the coding window.

If you had a long list of numbers to display, you don't really want to type them all out by hand! Instead, you can use a loop. Add the following loop to your code:

```
for (int i = 0; i != (lottery_numbers.Length); i++)
```

```
{
    listBox1.Items.Add( lottery_numbers[i] );
}
```

Now delete all of your list box lines. Your code should then look like this:

```
private void button1_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();

    int[] lottery_numbers;
    lottery_numbers = new int[4];

    lottery_numbers[0] = 1;
    lottery_numbers[1] = 2;
    lottery_numbers[2] = 3;
    lottery_numbers[3] = 4;

    for (int i = 0; i != (lottery_numbers.Length); i++)
    {
        listBox1.Items.Add(lottery_numbers[i]);
    }
}
```

When you run the programme, the numbers should display in the list box again. But how does it work?

The code works because the array index number is matching the loop variable number. Here are some images to show what's happening:

```
private void button1_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();

    int[] lottery_numbers;
    lottery_numbers = new int[4];

    lottery_numbers[0] = 1;
    lottery_numbers[1] = 2;
    lottery_numbers[2] = 3;
    lottery_numbers[3] = 4;

    for (int i = 0; i != (lottery_numbers.Length); i++)
    {
        listBox1.Items.Add(lottery_numbers[i]);
    }
}
```

Assign a value of zero to the loop variable called i

```

private void button1_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();

    int[] lottery_numbers;
    lottery_numbers = new int[4];

    lottery_numbers[0] = 1;
    lottery_numbers[1] = 2;
    lottery_numbers[2] = 3;
    lottery_numbers[3] = 4;

    for (int i = 0; i != (lottery_numbers.Length); i++)
    {
        listBox1.Items.Add(lottery_numbers[i]);
    }
}

```

Use the value in the **i variable as the index number of the array**

In the first image, we've highlighted the **int** variable we'll call **i**. This gets set to zero, which is the start of the loop. In the second image, we see the **i** variable again. This time, it is between the square brackets of the array name. The first time round the loop, the value in **i** is 0. The **i** variable gets 1 added to it each time round the loop. So the second time round, its value will be 1, the third time 2, etc. So this is happening:

| Loop | Array |
|-------------|-----------------------------|
| i = 0 | lottery_numbers[0] |
| i = 1 | lottery_numbers[1] |
| i = 2 | lottery_numbers[2] |
| i = 3 | lottery_numbers[3] |

The value in each position is then accessed, which for us was the numbers 1 to 4.

One thing to make note of is this part of the **for** loop:

i != (lottery_numbers.Length)

Length is a property of arrays that you can use. It refers to the number of items in your array. So we're saying, "Keep looping while the value in **i** does not equal The **Length** of the array".

Use a loop to assign values to an array

You can also use a loop to assign values to your arrays. In the code below, we're using a loop to assign values to our **lottery_numbers** array:

```

for (int i = 0; i != (lottery_numbers.Length); i++)
{
    lottery_numbers[i] = i + 1;
    listBox1.Items.Add(lottery_numbers[i]);
}

```

The only thing that has changed with our for loop is the addition of this line:

lottery_numbers[i] = i + 1;

The first time round the loop, the value in **i** will be zero. Which gives us this:

lottery_numbers[0] = 0 + 1;

The second time round the loop, the value in **i** will be 1. Which gives us this:

lottery_numbers[1] = 1 + 1;

But what we are doing is manipulating the index number (the one in square brackets). By using the value of a loop variable, it gives you a powerful way to assign values to arrays. Previously, we did this to assign 4 numbers to our array:

```

lottery_numbers[0] = 1;
lottery_numbers[1] = 2;
lottery_numbers[2] = 3;
lottery_numbers[3] = 4;

```

But if we need 49 numbers, that would be a lot of typing. Contrast that to the following code:

```

private void button1_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();

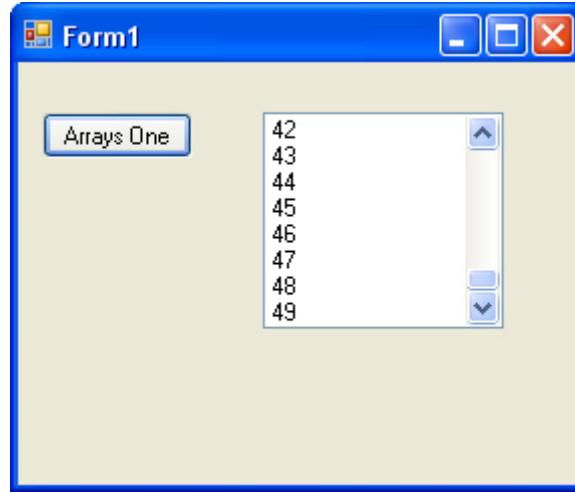
    int[] lottery_numbers;
    lottery_numbers = new int[49];

    for (int i = 0; i != (lottery_numbers.Length); i++)
    {
        lottery_numbers[i] = i + 1;
        listBox1.Items.Add(lottery_numbers[i]);
    }
}

```

Here, we've set up the array for 49 numbers. We've used a loop to assign the values 1 to 49 to each position in our array. So with one small change, we've saved ourselves an awful lot of typing!

Change your own code to match ours and try it out. When you click the button on your form, all 49 numbers should appear in the list box:



As an exercise, halt your programme and change the index number of the array from 49 to 1000. Run your programme and test it out. What you've done is to set up an array and fill it with a thousand values!

Set the size of an array at runtime

The size of an array refers to how many items it holds. You've seen that to set the size of an array, you do this:

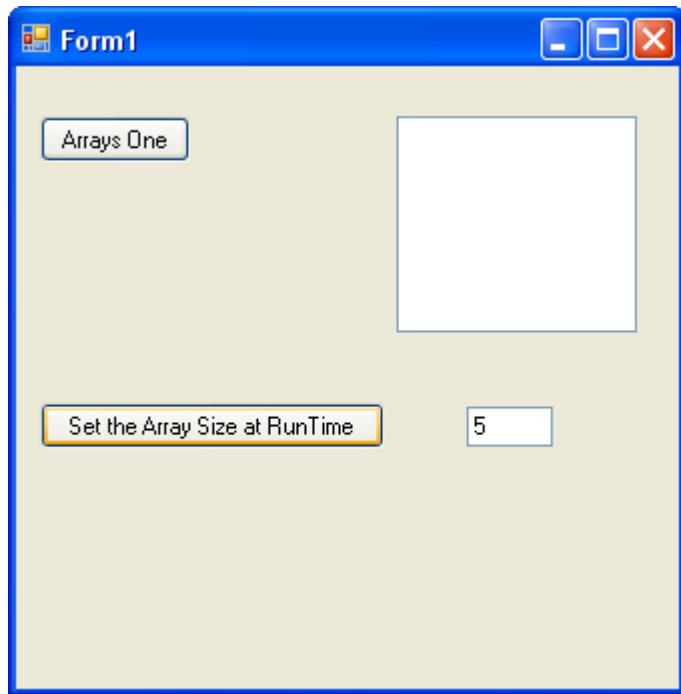
```
int[] someNumbers;  
someNumbers = new int[10];
```

Or this:

```
int[] someNumbers = new int[10];
```

But sometimes, you just don't know how big the array needs to be. Think of a programme that pulls information from a database. You want to loop round and check how many of your customers still owe you money. You've decided to hold all the data in an array. But how big does the array need to be? You can't set a fixed size before the programme runs, simply because the number of people owing you money could change. It could be low one month, and high the next.

To solve the problem, you can set the array size at runtime. This would mean, for example, setting the array size after a button is clicked. To see how it works, add another button to your form, along with a text box. Your form should then look something like this:



We've typed the number 5 in the text box, but you can have any number you like.

Double click your button and enter the following code:

```
int aNumber = int.Parse(textBox1.Text);

int[] arraySize;
arraySize = new int[aNumber];
```

The first line of the code gets the value from the text box and places into a variable we've called **aNumber**. The second line sets up an array as normal. But look at the third line:

```
arraySize = new int[aNumber];
```

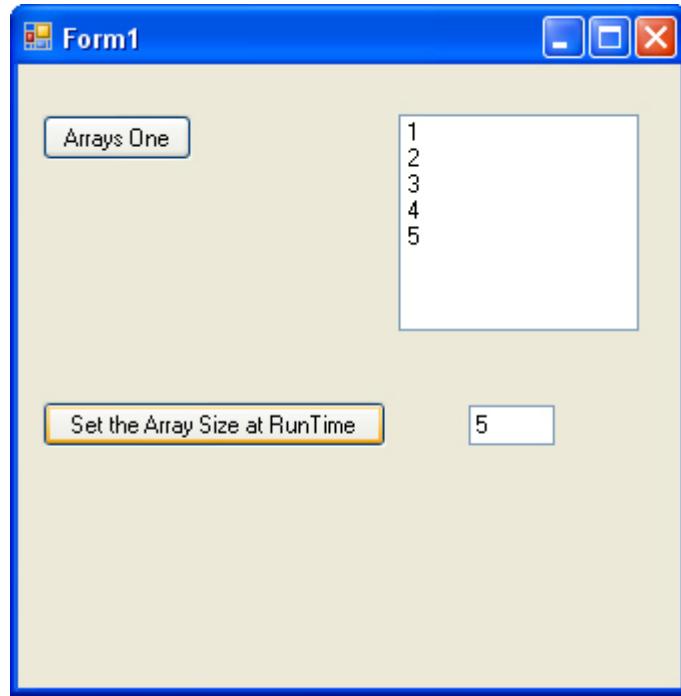
Now, the figure between the square brackets of **int** is not a number we've just typed. It's a variable name. Since the value of the variable is coming from the text box, the size of the array will be whatever number is typed in the text box.

Add the following loop to your code, which just assigns values to the array, and places them in the list box on your form:

```
for (int i = 0; i != (arraySize.Length); i++)
{
    arraySize[i] = i + 1;

    listBox1.Items.Add(arraySize[i]);
}
```

Run your programme and click your button. You should see this in your list box:



Now delete the 5 and type a different number. You should see the number from 1 to whatever number you've just typed.

This technique can be used whenever your programme needs to get its array size at runtime – assign to a variable, and use this variable between the square brackets of your array.

To get some more practice with arrays, add another button to your form. Set the text to be this:

Exercise: Times Table

Now double click the button and add the following code:

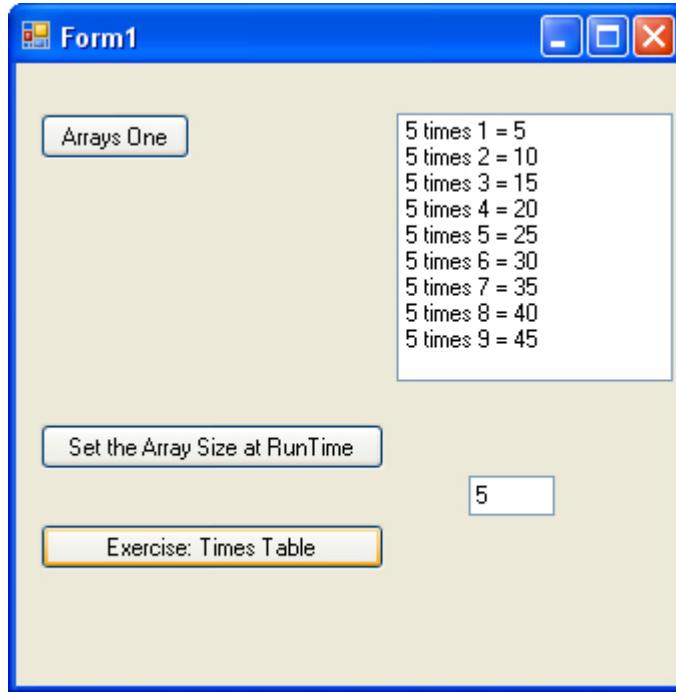
```
private void button3_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();

    int[] aryTimes;
    aryTimes = new int[10];

    int times = int.Parse(textBox1.Text);

    for (int i = 1; i != (aryTimes.Length); i++)
    {
        aryTimes[i] = i * times;
        listBox1.Items.Add(times + " times " + i + " = " + aryTimes[i]);
    }
}
```

When you run your programme and click the button, your form should look something like ours:



Notice that the 5 times table is displayed in the list box. Now try the following exercises:

Exercise K

The first item in the array, `arrayTimes[0]`, doesn't get used - why is this?

[Answer to Exercise K](#)

Exercise L

Amend the code so that the times table from 1 to 10 displays in the list box, not 1 to 9

[Answer to Exercise L](#)

Multi Dimensional Arrays

Your arrays don't have to be single lists of items, as though they were a column in a spreadsheet: you can have arrays with lots of columns and rows. These are called Multi-Dimensional arrays. For a 1-dimensional array, the ones you've been using, it would look like this:

Column 1

| | |
|-------------------------|----|
| Array Position 0 | 10 |
| Array Position 1 | 20 |
| Array Position 2 | 30 |

| | |
|-------------------------|----|
| Array Position 3 | 40 |
| Array Position 4 | 50 |

For a 2-dimensional array, it would look like this:

| | Column 0 | Column 1 | Column 2 |
|-------------------------|-----------------|-----------------|-----------------|
| Array Position 0 | 10 | 100 | 1000 |
| Array Position 1 | 20 | 200 | 2000 |
| Array Position 2 | 30 | 300 | 3000 |
| Array Position 3 | 40 | 400 | 4000 |
| Array Position 4 | 50 | 500 | 5000 |

So if you wanted to get at the value of 2000 in the table above, this would be at array position 1 in column 2 (1, 2). Likewise, the value 400 is at position 3, 1.

To set up a 2-dimensional array, you use a comma:

int[,] arrayTimes;

You then need a number either side of the comma:

arrayTimes = new int[5, 3];

The first digit is the number of Positions in the array; the second digit is the number of Columns in the array.

Filling up a 2-dimensional array can be quite tricky because you have to use loops inside of loops! Here's a programme that fills a 2-dimensional array with the values in the table above:

```
private void button5_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();

    int arrayRows = 5;
    int arrayCols = 3;

    int[,] arrayTimes;
    arrayTimes = new int[arrayRows, arrayCols];

    int mult = 0;

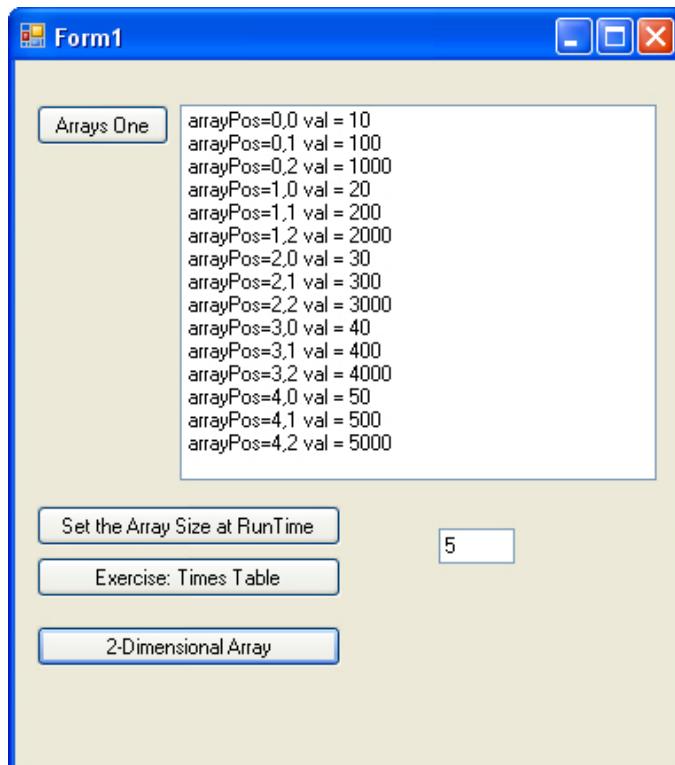
    for (int i = 0; i != arrayRows; i++)
    {
        mult = mult + 10;

        for (int j = 0; j != arrayCols; j++)
        {
            arrayTimes[i, j] = mult;
            mult = mult * 10;
        }

        mult = mult / 1000;
    }
}
```

Notice the two for loops in the code above, one inside of the other. The first loop is setting the value of the Rows (array Positions), and the second loop is setting the value of the Columns.

You don't have to understand the code, at this stage of your career! But see if you can puzzle it all out. For the adventurous, add another button to your form. Enter the code above. Now add a second double **for** loop, and print out the array values in your list box. See if you can get the same values as ours, in the form below:



This is a tough exercise, so give yourself a giant pat on the back, if you get there!

Arrays and Text

You've seen how to place numbers into an array. But you can also place strings of text.

Add a new button to your form, and set the Text property to **String Arrays**. Then double click your button to get at the code.

To place text into an array, you set up your array as you normally would, except you use the keyword **string** instead of **int**. Like this:

```
string[] arrayStrings;
arrayStrings = new string[5];
```

So the above code would set up an array that is going to hold strings of text. There will be five positions in the array.

Add the code above to your button.

To place values in the array, it's just normal variable assignment, with a pair of square brackets after the variable name:

```
arrayStrings[0] = "This";
arrayStrings[1] = "is";
arrayStrings[2] = "a";
arrayStrings[3] = "string";
arrayStrings[4] = "array";
```

Add the above to your code.

The foreach Loop

To access the values in each position of your array, you could use a **for** loop, as you have been doing. Like this:

```
for (int i = 0; i != (arrayStrings.Length); i++)
{
    listBox1.Items.Add( arrayStrings[i] );
}
```

But there's another type of loop that you haven't met yet called a **foreach** loop. This comes in handy when you're trying to access items in a collection, which you'll see how to do shortly. But add this to your code, instead of a **for** loop:

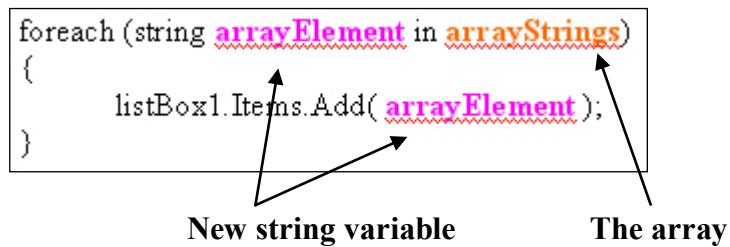
```
foreach ( string arrayElement in arrayStrings )
{
    listBox1.Items.Add( arrayElement );
}
```

Notice where all the keywords are in the loop code above, the ones in bold. You start with the **foreach** keyword, followed by a pair of round brackets. In between the round brackets, we have this:

string arrayElement in arrayStrings

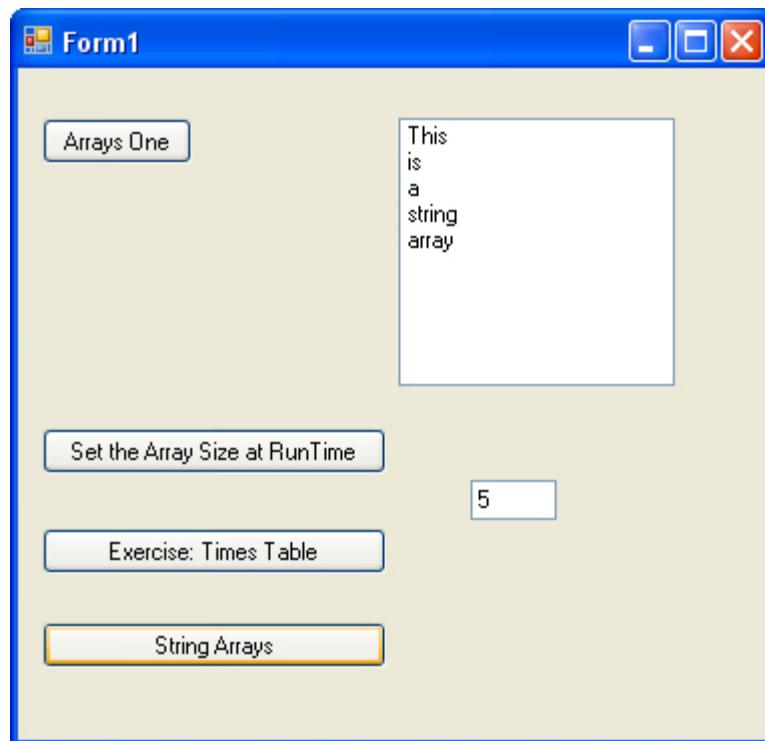
This is really two parts in one. In the first part, **string arrayElement**, you set up a new variable. The new variable will hold the elements (array values) from each position in your array. The second part, **in arrayStrings**, is where you tell C# the name of the array or collection you want to access. C# will then loop round all the positions in your array, and place the value at that position into your new variable (**arrayElement**, for us). You can then do something with the value in the new

variable. All we are doing in our loop is to display the value in a list box. Here's an image that may help you to understand what's going on:



Notice that for the array, you don't need the square brackets anymore. The neat thing about **foreach** loops is that you don't need to use index numbers, like you do with ordinary for loops.

Run your programme and click the button. The list box on your form should then look like ours:



The final part of this section deals with objects closely associated with arrays – collections.

Collections

Arrays are very useful for holding lots of values under the same name. But there is also something called a Collection that does a similar job. In fact, there's an inbuilt group of Classes in C# specifically for Collections. They can be quite powerful.

With an array, you store data of the same type. An array set up as **int** can hold numbers, but not letters. And an array set up as **string** can't hold numbers. So you can't do this, for example:

```
arrayPos[0] = 1;
arrayPos[1] = "two";
```

The first position holds a number and the second position holds text. C# won't let you do this in an array. But you can do it in a collection known as a Hashtable.

Collections can also make it easier to do things like sorting the data in your lists, deleting items, and adding more items. We'll start with the collection class called Lists.

Lists

You use a List when your collection may need items adding to it, deleting from it, or needs sorting. For example, suppose you are teacher with a class of ten children. You could keep a list of the children's names, add new students, delete ones who leave, and even sort them alphabetically! If you used a normal array, it would be difficult to do these things.

So start a new C# project. Add a button and a listbox to your new form. Double click the button to get at the coding window. Now have a look near the top and you'll see a list of **using** statements (lines 1 to 7 in the image below):

```
1| using System;
2| using System.Collections.Generic;
3| using System.ComponentModel;
4| using System.Data;
5| using System.Drawing;
6| using System.Text;
7| using System.Windows.Forms;
8|
9| namespace collections
10| {
11|     public partial class Form1 : Form
12|     {
13|         public Form1()
14|     }
```

The **using** statement that is needed for lists is the one that says **System.Collections.Generics**. If you can't see it as one of your **using** statements, add it yourself.

To set up a List, double click the button on your form to create a code stub. Now add the following line:

```
List<string> students = new List<string>();
```

You start with the word, **List**. Then comes a pair of pointy brackets. In between the pointy brackets you need a variable or object type. We've used **string**. What this tells C# to do is to set up a List that will hold string values. The list itself is called **students**. After an equal sign, we have the **new** keyword. Next comes the **List<string>** part again. This time, however, it's followed by a pair of round brackets, telling C# to create a new list of string values.

Your code window will look something like this:

```
namespace lists
{
    public partial class Form1 : Form
    {
        public Form1()
        {

            private void button1_Click(object sender, EventArgs e)
            {

                List<string> students = new List<string>();

            }
        }
    }
}
```

After setting up the List, you need to fill it with data. Add the following three lines to your code:

```
students.Add("Jenny");
students.Add("Peter");
students.Add("Mary Jane");
```

Your coding window will then look like this:

```
private void button1_Click(object sender, EventArgs e)
{

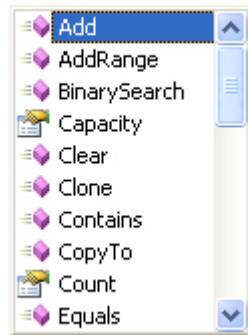
    List<string> students = new List<string>();

    students.Add("Jenny");
    students.Add("Peter");
    students.Add("Mary Jane");

}
```

After typing the name of your List (**students**, for us), you may have seen the C# IntelliSense list appear:

students.



This is a list of all the Methods and Properties that a List has. The one that you use to add items to your List is called, not surprisingly, **Add()**. Between the round brackets of Add(), you type the data that you want to add to your ArrayList:

```
students.Add("Jenny");
```

For every item in your collection, you need a new line that **Adds** items.

To access the items in your List, you can use a For Each loop. Add this to your button code:

```
foreach (string child in students)
{
    listBox1.Items.Add( child );
}
```

So we're looping round all the items in the List, and then adding them to the listBox.

We could have used an ordinary for loop for all this, but it's more awkward:

```
for (int i = 0; i < students.Count; i++)
{
    listBox1.Items.Add( students[i] );
}
```

Notice that the end condition is **students.Count**. **Count** is a property of Lists that tells you how many items is in it. Inside the for loop, we're using square brackets with the index number inside. This is just like the normal arrays you used earlier.

But if you want to loop round a collection, the above code is not the right choice. A better choice is a foreach loop.

A foreach loop ends when no more items in your collection or array are left to examine. Unlike a normal for loop, you don't have to tell C# when this is – it already knows what's in your collection, and is clever enough to bail out of the foreach loop by itself.

Adding Items to a List

You can add a new item to your List at any time. Here's an example to try:

```
private void button1_Click(object sender, EventArgs e)
{
    List<string> students = new List<string>();

    students.Add("Jenny");
    students.Add("Peter");
    students.Add("Mary Jane");

    foreach (string child in students)
    {
        listBox1.Items.Add(child);
    }

    students.Add("Azhar");
    listBox1.Items.Add("new item = " + students[3]);
}
```

So we're adding a fourth student to the List, Azhar, and then displaying the item in the listbox.

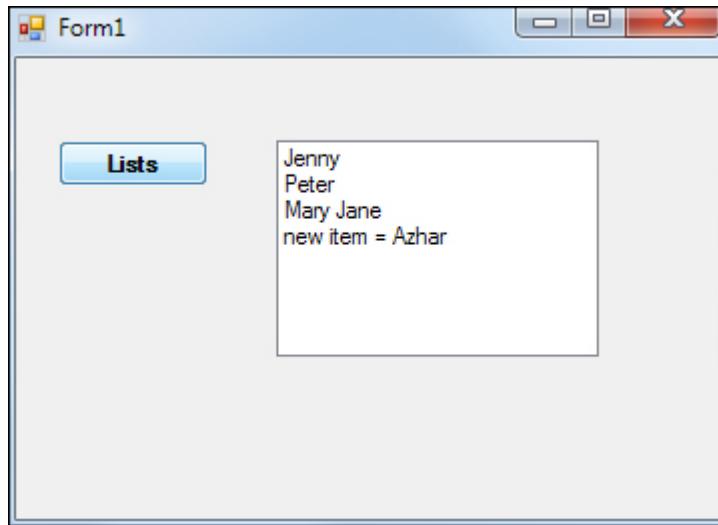
Notice that you can still access items in the list by using square brackets:

students[3]

If you want to use square brackets then just remember that the first item in the list will be item[0] and not item[1].

When you add items to a list with the Add method the new item gets added to the end of the list.

Add the new code to your button. Run your programme and click your button. Your form will look something like this:



Sorting a List

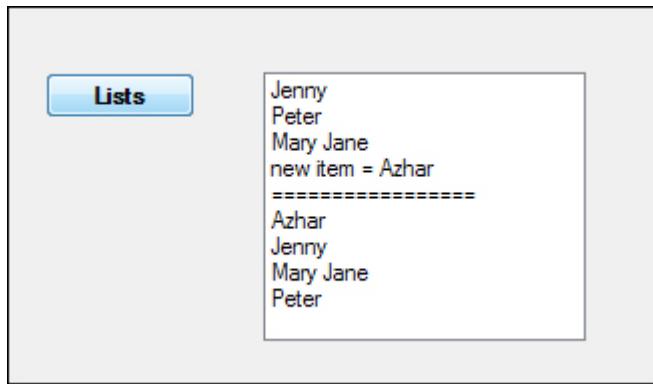
Sorting a List alphabetically is quite straightforward. You just use the **Sort** Method. Like this:

```
students.Sort();
```

And here's some code to try out. The new lines should be added at the end of your current code:

```
students.Sort();  
  
listBox1.Items.Add("=====");  
  
foreach (string child in students)  
{  
    listBox1.Items.Add(child);  
}
```

Run your programme and try it out. Here's what your listbox will look like after the button is clicked:



As you can see, the items have been sorted alphabetically, in ascending order. You can have a descending sort, if you prefer. One way to do this is with the **Reverse** method:

```
students.Reverse();
```

No extra coding is needed!

Removing items from a List

To remove an item from your list, you can either use the **Remove** or the **RemoveRange** methods. The Remove method deletes single items from the List. You use it like this:

```
students.Remove("Peter");
```

In between the round brackets of Remove, you simply type the item you want to remove.

If you want to remove more than one item, use **RemoveRange**. Like this:

```
students.RemoveRange(0, 2);
```

The first number between the round brackets of **RemoveRange** is where in your list you want to start. The second number is how many items you want to remove. Here's some code to try at the end of your button:

```
students.RemoveRange(0, 2);

listBox1.Items.Add("=====");
foreach (string child in students)
{
    listBox1.Items.Add(child);
}
```

But that's enough of Lists. There's lots more that you can do with them, and they are worth researching further.

The final Collection we'll look at is called a Hashtable.

Hashtables

You use a Hashtable when you want to store information based on Key/Value pairs. For example, the name of a student AND the score in an exam. This allows you to mix text and numbers. (In other programmes, Hashtables are known as associative arrays.)

Add a new button to your form, and double click it to get at the code. Now have a look at the **using** statements at the top. Add the following to the end of the list:

```
using System.Collections;
```

You already have a **System.Collections.Generic** statement, but a Hastable is not part of this collection – it's part of the normal Collections namespace.

Click inside the code stub for you button. You setting up the Hashtable like this:

```
Hashtable students = new Hashtable();
```

This creates a new object called **students**. It's going to be a Hashtable object.

There are two ways you can add data to your Hashtable. Like this:

```
students["Jenny"] = 87;  

students["Peter"] = "No Score";  

students["Mary Jane"] = 64;  

students["Azhar"] = 79;
```

Or like this:

```
students.Add("Jenny", 87);  

students.Add("Peter", "No Score");  

students.Add("Mary Jane", 64);  

students.Add("Azhar", 79);
```

The first method uses a pair of square brackets:

```
students["Jenny"] = 87;
```

In between the square brackets, you type what's known as the Key. So this particular entry in the Hashtable is called “Jenny”. After an equals sign, you then

type the Value that this Key will hold. Notice that three of the entries are number values, and one (Peter) is text.

The second way to store values in a Hashtable is to use the **Add** Method:

```
students.Add("Jenny", 87);
```

In between the round brackets of **Add()**, you first type the Key name. After a comma, you type the Value for that Key.

There is a difference between the two. If you use the Add method, you can't have duplicate Key names. But you can if you use the square brackets. So this will get you an error:

```
students.Add("Jenny", 87);  
students.Add("Jenny", 35);
```

But this won't:

```
students["Jenny"] = 87;  
students["Jenny"] = 35;
```

To try Hashtables out for yourself, add the following code to your button:

```
Hashtable students = new Hashtable();  
  
students["Jenny"] = 87;  
students["Peter"] = "No Score";  
students["Mary Jane"] = 64;  
students["Azhar"] = 79;  
  
foreach (DictionaryEntry child in students)  
{  
    listBox1.Items.Add("student: " + child.Key + " , Score: " + child.Value);  
}
```

Before running the code, have a look at the foreach loop. Inside of the round brackets, we have this:

DictionaryEntry child

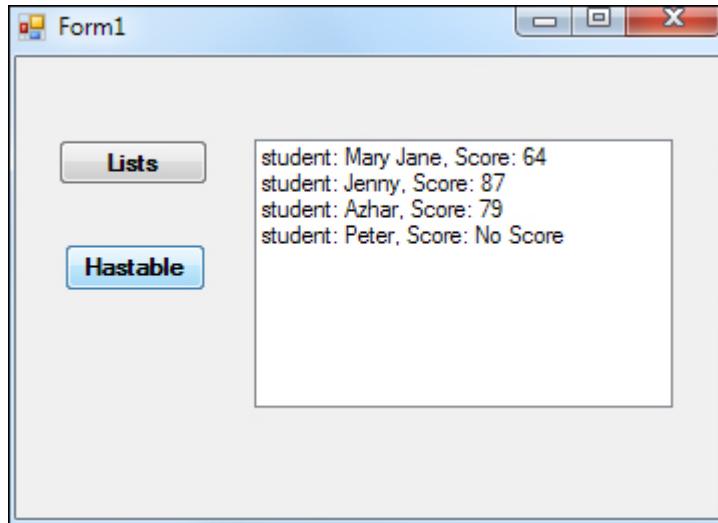
This sets up a variable called **child**. But note the type of variable it is: a **DictionaryEntry**. C# uses an object of this type when using the foreach loop with Hashtables. That's because it automatically returns both the Key and the Value.

Notice, too, what we have between the round brackets of the listbox's **Add** method:

```
"student: " + child.Key + " , Score: " + child.Value
```

The bold parts are the important parts. After typing the name of your variable (**child**, for us) and a full stop, the IntelliSense list will appear. Key is a property that returns the name of your Key, and Value is a property that returns whatever you placed inside of that Key.

Run your programme and click your button. The form on your listbox will then look like this:



But just like the List, you can Add new items, and remove old ones. To **Remove** an item, you do it like this:

```
students.Remove("Peter");
```

So you refer to the Key name, and not the Value, when you use the Remove method.

Enumerations

You can create your own collections of things with something called **Enumeration**. Suppose you wanted to set up a list of subjects that students can study, but don't want to keep typing the list out all the time. Instead, you can set up an **Enumerated** list. Here's how.

Add a new button to the form and set the text property as **Enumeration**. Double click your button to get at the coding window. Now add the enum part below to your code:

```

public partial class Form1 : Form
{
    public Form1() { }

    enum Subjects { English, IT, Science, Design, Math };

    private void button1_Click(object sender, EventArgs e)
    {
    }
}

```

You start with the word **enum**. After a space, you type a name for your enumerated list. In between a pair of curly brackets, you type the list itself. We've added five subjects to our lists; English, IT, Science, Design, Math.

To use your enumerated list, click inside of your button code. Add the following line:

Subjects newSubject = Subjects.Science;

So you type the name of your enumerated list followed by a space. After coming up with a variable name, type an equals sign. Type the name of your list again, followed by a dot. You will then see the items in your list:

```

enum Subjects { English, IT, Science, Design, Math };

private void button1_Click(object sender, EventArgs e)
{
    Subjects newSubject = Subjects.|
```



Select one of the items from your list, and end the line with a semicolon.

What you have done is to set up your own value type, with its own name (Subjects). The values in your type are the ones you added between the curly brackets. They also have an underlying number. So the first item is 0, the second item is 1, the third 2, etc.

To display it in a message box as a string, the code would be this:

```
MessageBox.Show( newSubject.ToString() );
```

It needs converting because it's an enumerated type – it's not a string.

If you want to get at the underlying number, you would do the conversion like this:

```
Subjects newSubject = Subjects.Science;  
int enumNumber = (int)newSubject;
```

In other words, convert to **int** in the usual way, by putting **(int)** before what it is you want to convert. You could then loop round and do some checking. Something like this:

```
for (int i = 0; i < 4; i++)  
{  
    if (i == 2)  
    {  
        MessageBox.Show("You're taking Science");  
    }  
}
```

Enumerated lists are good for when you have a list of custom items and don't want to use an array.

And that ends the section on arrays and collections. There's still an awful lot we haven't yet covered on the subject, especially for Collections. But it's enough to be going on with, for a beginner! In the next part, we'll have a close look at Strings.

String Manipulation in C#

Quite often, strings of text need manipulating. Data from a textbox need to be tested and checked for things like blank strings, capital letters, extra spaces, incorrect formats, and a whole lot more besides. Data from text files often needs to be chopped and parsed before doing something with it, and the information you get from and put into databases routinely needs to be examined and worked on. All of this comes under the general heading of String Manipulation.

Let's go through a few of the things that will help you deal with strings of text.

String Variables

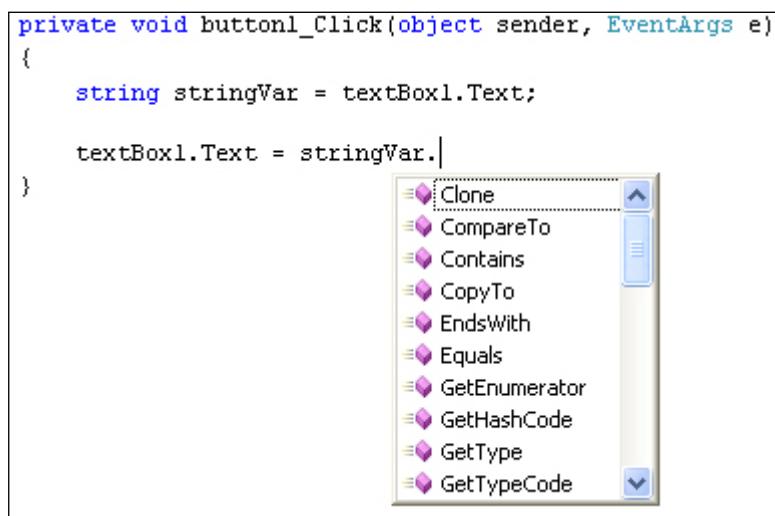
You've already worked with string variables a lot in this book. But there's a lot more to them than meets the eye. Strings come with their own Methods and Properties that you can make use of. To see which Methods and Properties are available, start a new C# Windows Application. Add a button and a textbox to your form. For the textbox, change the Text property to "some text" (make sure the text is in lowercase). Now double click your button to get at the coding window. Then enter the following string declaration

```
string stringVar = textBox1.Text;
```

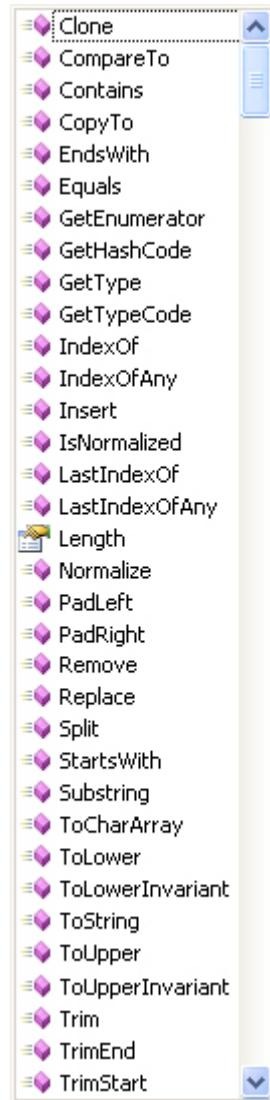
On a new line, type the following:

```
textBox1.Text = stringVar.
```

As soon as you type the full stop at the end, you'll see the IntelliSense list appear:



IntelliSense is showing you a list of Methods and Properties that are available for this string object you have called **stringVar**. Here's a fuller list:



There's actually one Property on the list. But it's one you use a lot, and we'll see it in action later.

Most of the Methods on the list you won't use at all, and a lot are just plain baffling! Some are quite obvious in what they do.

Select **ToUpper** from the list by double clicking it. Because it's a Method, you need some round brackets. Type a left round bracket and you'll see a yellow box appear, giving you the available options for this Method. For the **ToUpper** Method, there are only two options available:

```

private void button1_Click(object sender, EventArgs e)
{
    string stringVar = textBox1.Text;

    textBox1.Text = stringVar.ToUpper();
}
1 of 2 string ToUpper()
Returns a copy of this System.String converted to uppercase, using the casing rules of the current culture.

```

You can press the Down arrow on your keyboard to see the other one. But the first one, 1 of 2, is the one we need. As the tool tip is telling you, this Method converts the string to upper case. (The current culture it is talking about is which language you're typing in: a symbolic language like Chinese or Japanese will have different grammatical rules than English.)

The round brackets of the Method are empty, meaning it doesn't take any arguments. So just type the right round bracket, followed by a semicolon to end the line. Your code should look like this:

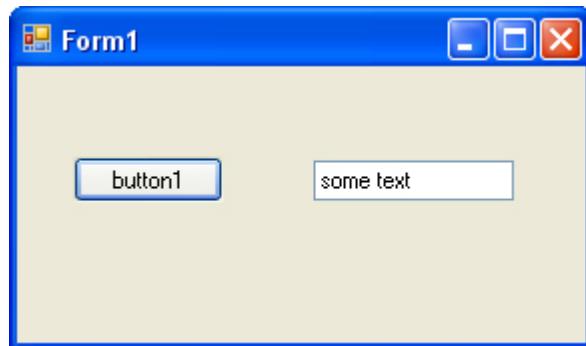
```

private void button1_Click(object sender, EventArgs e)
{
    string stringVar = textBox1.Text;

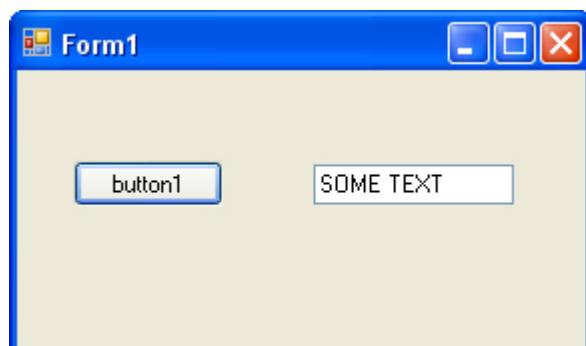
    textBox1.Text = stringVar.ToUpper();
}

```

Now run your programme. When the form starts it will look like this:



After you click the button, C# runs the **ToUpper** Method and converts the text in the text box to uppercase:



Another Method that changes case is the **ToLower** Method. This is the opposite of **ToUpper**, and is used in the same way.

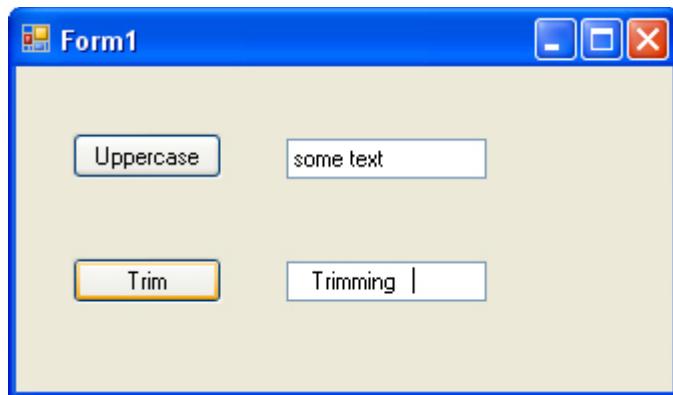
Trimming Unwanted Characters

If you have another look at the Method list, you'll see that there are three that deal with Trimming: Trim, TrimEnd and TrimStart. These methods can be used to Trim unwanted characters from strings.

Add another button to your form. You can change the Text property of your buttons. Enter the text "Uppercase" for the first one. For the new button, enter Trim for the text property. Add another text box below the first one, and set the Text property as follows:

“ Trimming ”

Leave out the double quotes but tap the spacebar on your keyboard three times before you type the text. At the end of the text, tap the spacebar three more times. Your Form should then look like this:



The line in the second text box is where the cursor is.

Now double click your second button to get at the code. We can count the number of characters a string has with the Length property. Enter the following code for your button:

```
private void button2_Click(object sender, EventArgs e)
{
    string stringTrim = textBox2.Text;

    int stringLength = stringTrim.Length;

    MessageBox.Show(stringLength.ToString());
}
```

The first line just gets the text from the text box and puts it into a variable called **stringTrim**. Have a look at the second line, though:

```
int stringLength = stringTrim.Length;
```

We've set up a new integer variable called **stringLength**. To get the length of a string, type a dot after your string variable name. From the IntelliSense list, select the Length property. Note that you don't need any round brackets, because it's a property not a method. The Length of a string, by the way, refers to how many characters are in the string.

The third line uses a MessageBox to display the result:

```
MessageBox.Show( stringLength.ToString() );
```

You've seen the **ToString** method before. This can be used to convert numbers to a string of text. So "10" instead of 10. (The double quotes mean it's text. Without the quotes, it's a number. The variable called **stringLength** will hold a number.)

Run your programme and click the Trim button on your form. The message box should display an answer of 14. The word "Trimming", however, only has 8 characters in it. The other 6 are the three spaces we put at the beginning and end of the word.

To get rid of space at the beginning and end of text, you can use the **Trim** method. Add the following line of code to your button:

```
private void button2_Click(object sender, EventArgs e)
{
    string stringTrim = textBox2.Text;

    stringTrim = stringTrim.Trim();

    int stringLength = stringTrim.Length;

    MessageBox.Show( stringLength.ToString() );
}
```

The code to add is highlighted, in the image above. It's this:

```
stringTrim = stringTrim.Trim();
```

So after the dot of the **stringTrim** variable, you select the **Trim** method from the IntelliSense list, followed by a pair of empty round brackets. Run your programme and click the button again. You should find that the length is now 8. So **Trim** has trimmed the blank spaces from the beginning and the end of our word.

If you only wanted to trim the blank spaces at the end of the word, or just the blank spaces at the beginning of the word, you can use **TrimEnd** and **TrimStart**:

```
stringTrim = stringTrim.TrimStart( null );
```

TrimStart and TrimEnd are supposed to take a character array as a parameter. If you type the keyword **null** instead, it will trim the white space (blank characters).

Just as a reference for you, here's some code that strips unwanted hyphens off the end of a string:

```
private void button3_Click(object sender, EventArgs e)
{
    string stringTrim = "Some Text---";

    MessageBox.Show(stringTrim);

    char[] trimChars = {'-'};

    stringTrim = stringTrim.TrimEnd(trimChars);

    MessageBox.Show(stringTrim);
}
```

The **trimChar** line is a character array (**char[]**) with the hyphen in between curly brackets. This is then handed to the TrimEnd method as a parameter.

The Contains Method

The **contains** method can be used if you want to check if a string contains certain characters. It's fairly simple to use. Here's an example:

```
private void button4_Click(object sender, EventArgs e)
{
    string stringVar = "Some Text---";

    if (stringVar.Contains("-"))
    {
        MessageBox.Show("TRUE");

        //CALL HYPHEN REMOVE METHOD HERE
    }
}
```

After the contains method, you type a pair of round brackets. In between the round brackets, you type the text you're checking for. In our code, we're using an if statement. If it's true that the string contains a “-” character, then some code can be executed.

The IndexOf Method

The IndexOf method can be used to check if one character is inside of another. For example, suppose you want to check an email address to see if it contains the @ character. If it doesn't, you can tell the user that it's an invalid email address.

Add a new button and a new text box to your form. For the Text property of the text box, enter an email address, complete with @ sign. Double click your button to get at the code. Enter the following:

```
private void button5_Click(object sender, EventArgs e)
{
    string stringEmail = textBox3.Text;

    int result = stringEmail.IndexOf("@");

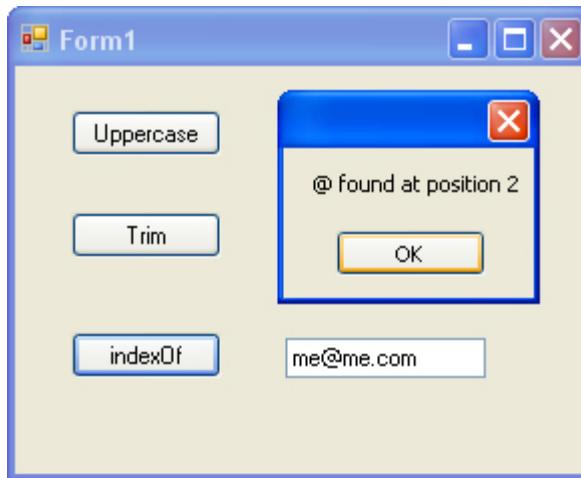
    if (result == -1)
    {
        MessageBox.Show("Invalid Email Address");
    }
    else
    {
        MessageBox.Show("@ found at position " + result.ToString());
    }
}
```

The first thing to examine is how IndexOf works. Here's the line of code:

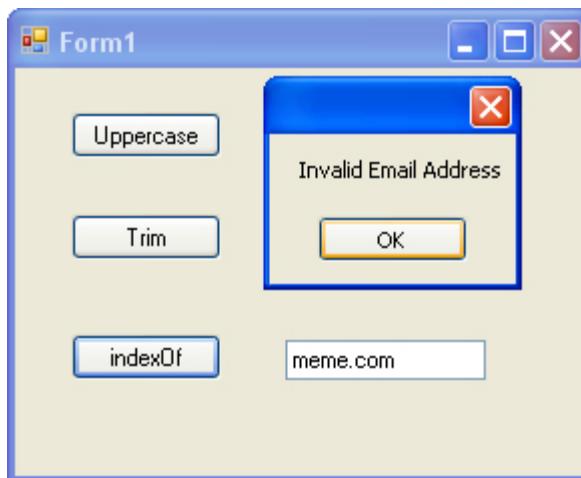
```
int result = stringEmail.IndexOf("@");
```

The IndexOf method returns an integer. This number is the character's position in the word you're trying to check. In the code above, we want to check the word that's inside of the variable we've called **stringEmail**. We want to see if it contains the "@" character. This goes between the round brackets of **IndexOf**. If C# finds the character, it will tell you where it was (at position number 3 in the word , for example). This number is then stored inside of the **int** variable we've called **result**. If the character you're looking for can't be found, IndexOf will return a value of -1 (minus 1). The IF statement in our code checks the value of the result variable, to see what's inside of it. If it's -1 display an " Invalid Email Address message"; if it's not -1 a different message is displayed.

Run your programme and click the button. Here's the form with an @ character in the text box:



And here's what happens when we delete the @ character from the text box:



Note that the first message box displays “@ found at position 2”. If you look at the email address in our text box, however, it's **me@me.com**. So you might be thinking that the @ character is at position 3, not 2. If C# were to start counting at 1, you'd be right. But it doesn't. When you use the IndexOf method, the count starts at zero.

You can also specify a start position and a character count for a search. This is useful if you want to do things like checking a longer string and counting how many occurrences there are of a particular character or characters. Or if you want a simple check to see if, say, a website entered in a text box on your form begins with **http://www**. Here's some code:

```

private void button6_Click(object sender, EventArgs e)
{
    string webAddress = "http://www.homeandlearn.co.uk";
    string checkWebAddress = "http://www";

    int start = 0;
    int numOfChars = 10;

    int result = webAddress.IndexOf(checkWebAddress, start, numOfChars);

    if (result != -1)
    {
        MessageBox.Show("Address OK");
    }
    else
    {
        MessageBox.Show("Address Bad");
    }
}

```

Have a look at this part of the highlighted line:

webAddress.IndexOf(checkWebAddress, start, numOfChars)

This time, we have three parameters inside of the round brackets of `IndexOf`. The first one is the string we want to check (`checkWebAddress`). Then we have `start`, and `numOfChars`. The `start` variable is where in your full string (`webAddress`) you want to start checking. The third parameter, `numOfChars`, is the number of characters you want to check from that starting position. In our code, the `start` is 0 and the number of characters is 10.

And finally, for `IndexOf`, here's some code that checks a long string of text and counts how many times the word `true` appears:

```

private void button3_Click(object sender, EventArgs e)
{
    string someText = "Is it true that true evaluates as true?";

    int strLen = someText.Length;
    int result = 0;
    int counter = 0;
    int foundLen = 0;

    for (int i = 0; i < strLen; i++)
    {
        result = someText.IndexOf("true", foundLen, strLen - foundLen);

        if (result != -1)
        {
            foundLen = result + 1;
            counter++;
        }
    }

    MessageBox.Show("true found " + counter.ToString() + " times");
}

```

The code is a bit complex, so don't worry if you don't understand it all. But it's just using IndexOf with three parameters: the word to search for, a starting position, and how many characters you want to check. The start position changes when the word is found; and the number of characters to count shrinks as you move through the word.

The Insert Method

The Insert method is a lot easier to use than IndexOf. It is used, not surprisingly, to insert characters into a string of text. You use it like this:

```
string someText = "Some Text";  
  
someText = someText.Insert( 5, "More " );
```

In between the round brackets of Insert, you need two things: A position in your text, and the text you want to insert. A comma separates the two. In our code above, the position that we want to insert the new text is where the T of "Text" currently is. This is the fifth character in the string (the count starts at zero). The text that we want to Insert is the word "More".

Exercise

Test out the code above, for the Insert() method. Have one message box to display the old text, and a second message box to display the new text.

The PadLeft and PadRight Methods

The PadLeft and PadRight methods can also be used to insert characters. But these are used to add characters to the beginning and end of your text. As an example, add a new button to your form, and a new text box. For the text box, set the Text property to "Pad Left". Double click your button and enter the following code:

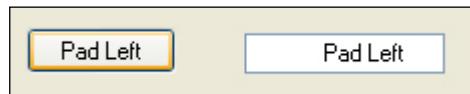
```
string paddingLeft = textBox5.Text;  
  
paddingLeft = paddingLeft.PadLeft(20);  
  
textBox5.Text = paddingLeft;
```

The PadLeft and PadRight methods can take 1 or two parameters. We're just using 1. This will insert blank space characters to the start of the string. The total number of characters will be 20. So, if you have four characters in the text box, PadLeft(20) will add 16 blank spaces, making a total of 20 characters in the text box after the button is clicked.

Run your programme and test it out. Type the text "Pad Left" in the text box. Your text box should look like this before the button is clicked:



And it will look like this after you click the button:



If you don't want to pad with blank spaces, you can use the second parameter. This is the character you want to pad with:

```
paddingLeft = paddingLeft.PadLeft(20, '*');
```

In the code above, we're telling C# to pad with the asterisk character, instead of the default blank spaces. (Note the use of the single quotes surrounding the * character. C# doesn't seem to like you using double quotes, if the type is **char**.)

If you change your code to the one above, then click the button on your form, the result will be this:



If you want to add characters to the end of the string, use PadRight instead of PadLeft.

The Remove Method

As its name suggests, this method is used to Remove characters from a string of text. Here's a simple example of its use:

```
string oldString = "some text text text";  
  
MessageBox.Show(oldString);  
  
string newString = oldString.Remove(10, 9);  
  
MessageBox.Show(newString);
```

Remove takes two parameters. The first one is what position in your string you want to start at. (The count starts at zero.) The second parameter is how many characters you want to delete, starting from the position you specified. Add another button to your form and try out the code above.

The Replace Method

The Replace method, you won't be surprised to hear, can be used to replace characters in your string. Here's an example of its use:

```
string spellingError = "mistak";  
  
spellingError = spellingError.Replace(spellingError, "mistake");
```

The Replace method takes two parameters, the old word and the new word. In the code above, we're replacing "mistak" with "mistake".

The Substring Method

The Substring method is used to grab characters from a string of text. For example, suppose you were testing an email address. You want to test the last four characters to see they are **.com**. You can use Substring to return just those four characters, and see what's in them. (You can also use IndexOf to achieve the same result.)

Substring has this syntax:

the_word.Substring(start_position)

So the word you want to grab characters from goes first, followed by the Substring method. In between the round brackets, you have to tell C# where in the word to start grabbing characters from.

But Substring can also take a second parameter:

the_word.Substring(start_position, num_of_chars_to_grab)

The second parameter is how many characters you want to grab. If you leave this out, C# will grab all the characters to the end of your word. Here's some code to try, with a new button:

```
private void button11_Click(object sender, EventArgs e)  
{  
    string emailAddress = "me@me.com";  
    string result = "";  
  
    result = emailAddress.Substring(5, 4);  
  
    if (result == ".com")  
    {  
        MessageBox.Show("Email Address OK");  
    }  
    else  
    {  
        MessageBox.Show("Bad Email Address");  
    }  
}
```

We're using Substring with two parameters (5, 4). But since we're grabbing to the end of the word, we could have left out the 4 at the end.

To test it out, change the **me@me.com** to **me@me.con**. Run your programme and you should see "Bad Email Address". Change it back and the email address will be OK.

Exercise M

Use Substring to check that an email address ends in .co.uk. For the email address to check, use **enquiry@me.co.uk**.

[Answer to Exercise M](#)

The Split Method

The Split method is used to split a string of text and put the words into an array. For example, you could grab a line of text from a text file. Each position in the array would then hold a word from the line of text. An example may clear things up.

Add another button to your form. Double click the button to get at the code, and add the following:

```
private void button12_Click(object sender, EventArgs e)
{
    string lineOfText = "item1, item2, item3";

    string[] wordArray = lineOfText.Split(',');
    MessageBox.Show( wordArray[0] );
    MessageBox.Show( wordArray[1] );
    MessageBox.Show( wordArray[2] );
}
```

Run the programme and click your button. You should see each word from the line of text display.

In the first line of the code, we're setting up a string with three items in it. Each item is separated by a comma. (Comma separated files from software like Excel are quite common, and so too is parsing each line of text.)

For the second line, we have this:

```
string[] wordArray = lineOfText.Split( ',' );
```

The first part sets up a string array that we've called **wordArray**. After the equals sign, we have this:

```
lineOfText.Split( ',' );
```

The variable called **lineOfText** is obviously the line of text we want to examine. For the round brackets of Split, we've typed a comma surrounded by single quotes. That's because C# needs to know what character in your line of text you are using to separate the words. This is known as the delimiter. If our line of text were this instead:

```
string lineOfText = "item1 item2 item3";
```

we'd use a blank space as a delimiter. Like this:

```
string[] wordArray = lineOfText.Split( ' ' );
```

This time, we've typed a blank space between the single quotes.

But C# will split the line, and put each part into the array we've set up. (It won't include the delimiter.) For our line of text we only have three words. So the Message box in our code displays what is at position 0, position 1, and position 2 in our array.

If you don't know how many positions there are in the array (if you have lines of text that vary in size, for example), you can loop through each position:

```
foreach (string s in wordArray)
{
    MessageBox.Show( s );
}
```

The Split method can take other parameters, and get a bit complex. So we'll leave it there in this beginner's book!

The Join Method

You can join the pieces of your arrays back together again. Join, however, is not a method available to ordinary strings. Instead, you can access it through the String class. Like this:

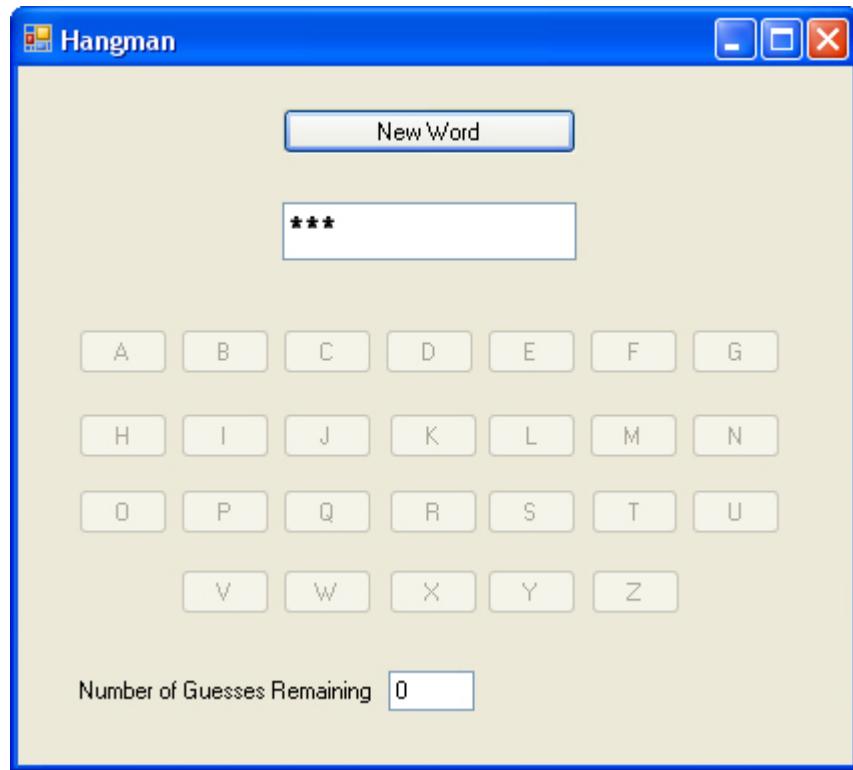
```
private void button13_Click(object sender, EventArgs e)
{
    string lineOfText = "item1, item2, item3";
    string[] wordArray = lineOfText.Split(',');
    string joinedText;
    joinedText = String.Join("-", wordArray);
    MessageBox.Show(joinedText);
}
```

In the code above, we've used **Split** to split the line of text and put the words into an array. We've then used **Join** to create a single line of text again. This time, though, the words are separated with hyphens and not commas.

To use **Join**, first type the word **String** (with a capital letter). After a dot, you should then see the **Join** method appear on the IntelliSense list. In between the round brackets of **Join**, you first need the character you want to use as a delimiter. Note that this is surrounded by double quotes. If you use single quotes, C# will think it is the **char** variable type. But you need to use the **string** variable type, so you'll get an error. After a comma, you type the name of the array you want to **Join** together

A C# .NET Hangman Game

To put all that theory into practice, we've developed a simple word-guessing game, popularly known as hangman in the UK. The project is amongst the files you downloaded at the start of the book. Look for the folder called **Hangman**. Inside of this, you'll find a **hangman.csproj**. Open up this project in your C# .NET software. Run the programme and you should see this:



After clicking the New Word button, you click on letters of the alphabet. If the word contains the letter you clicked on, it appears in the word. If you guess incorrectly then you lose a life. The game is over when you either guess the word, or run out of lives.

Our version contains words of only three letters, so that you can see what's going on. There are also message boxes, used to display the words. This is for testing purposes.

Open up the code for the project and you'll see that it is heavily commented. We won't go through the code step-by-step here, but examine the code and the comments for yourself. Pay particular attention to these C# string methods:

IndexOf
Insert
Remove
Substring
ToUpper

If you are unsure about any of them, go back a few pages and look at the explanation in this book.

As you're going through the code, though, bear in mind that what the programme is trying to do is manipulate strings and characters, using as many inbuilt C# methods as possible. This is the kind of manipulation that you need to be able to do in your own programming.

Once you've got a good idea of how the programme works, try this exercise.

Exercise

The game only uses words of three letters, at the moment. There are ten words in all. Amend the code to use words of nine letters. You can, of course, use your own. But here are 10 nine letter words for you, if you're stuck. They are all countries:

Argentina
Australia
Greenland
Guatemala
Indonesia
Lithuania
Macedonia
Mauritius
Nicaragua
Venezuela

For this exercise, it's not just a question of changing the words in the array. Can you see what else you need to change?

LINQ

LINQ stand for Language Integrated Query. As its name suggests, it's a way to query data. The kind of data you can use LINQ with includes collections, arrays, XML files, databases - in fact, just about any kind of data. (Technically, any data type that implements something called an `IEnumerable<T>` object.) LINQ is a huge subject and we can only scratch the surface here. But it's well worth investigating further. Let's take a brief look, though.

LINQ Methods and Properties

The first thing we'll look at is the simple methods and properties you can use with LINQ. We'll use these with a basic array of numbers. The LINQ methods we're going to look at are the following:

**SUM
MIN
MAX
AVERAGE
CONTAINS
ELEMENTAT
FIRST
LAST
DISTINCT
COUNT
TAKE
INTERSECT**

So start a new project for this. Add a button and a listbox to your form. Double click your button to open a code window. Now set up the following array of numbers:

```
int[] numberList = new int[5] { 1, 2, 3, 4, 5};
```

Now that we have an array to work with, add some information to your list box with the following:

```
listBox1.Items.Add("ARRAY ITEMS: 1, 2, 3, 4, 5");  
listBox1.Items.Add("=====");
```

This is just so that we can have dividing lines between each LINQ method we're going to use.

The SUM Method

The first LINQ query method to examine is SUM. Add the following line to your code:

```
int listTotal = numberList.Sum();
```

We've set up a variable called **listTotal**. It's of type **int** because SUM will return an integer total (it's going to be returning the total from the integer array we set up).

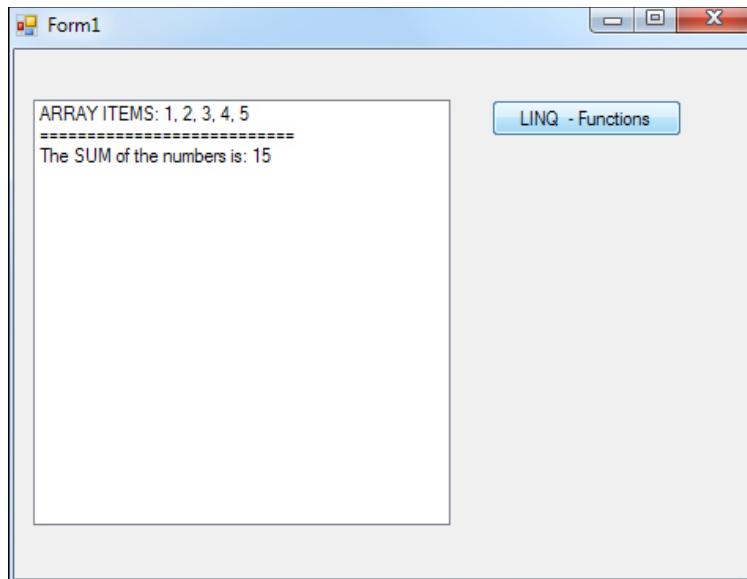
After an equal sign, we've typed the name of the array, **numberList**. Next comes a dot. After the dot, you type the name of the LINQ method you want to use, adding a pair of round brackets after its name. The one we've used is called **Sum**. This method will add up all the numbers in the array. However, the method won't be executed straightaway. All you're doing here is creating a query definition. The method is actually executed on the next line of our code.

To see it in action, add the following line:

```
listBox1.Items.Add("The SUM of the numbers is: " + listTotal);
```

It's only when you're doing something with a LINQ method or a LINQ query that it gets executed. We're doing something when we add the variable called **listTotal** to the listbox. When C# comes across **listTotal**, it then executes the SUM method.

Run your programme and test it out. When you click your button, your listbox will look like this:



So all the numbers in the array have been added together, just by using a simple LINQ method .

Minimum and Maximum

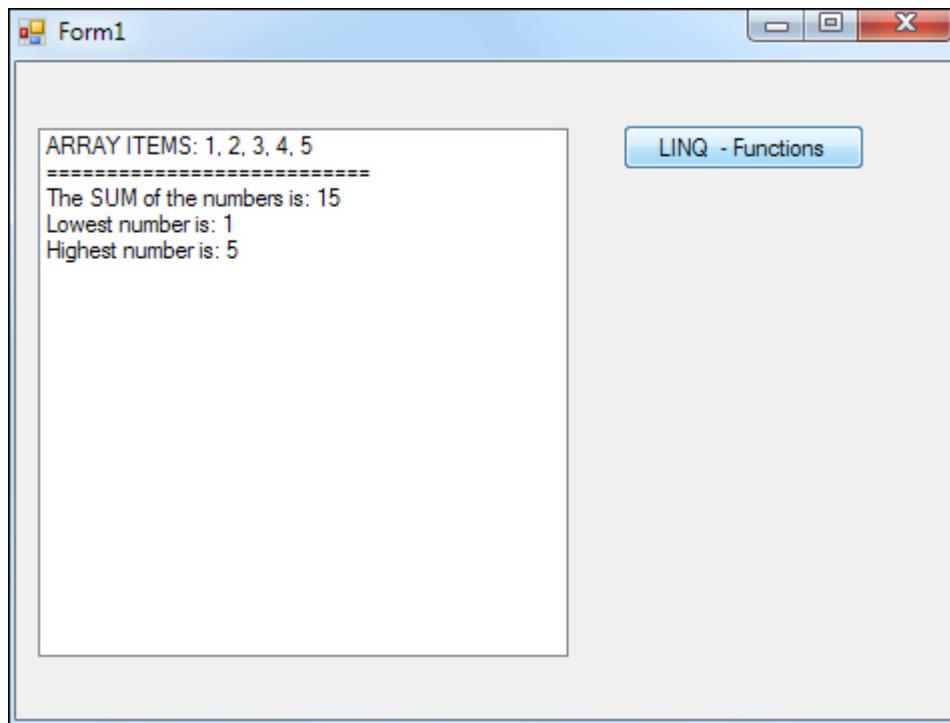
If you need to, you can get the lowest number in your array, or the largest number. The methods **MIN** and **MAX** are used for this. Add the following to your code:

```
int lowestNumber = numberList.Min();
listBox1.Items.Add("Lowest number is: " + lowestNumber);

int highestNumber = numberList.Max();
listBox1.Items.Add("Highest number is: " + highestNumber);
```

We've set up two new variables, one called **lowestNumber** and one called **highestNumber**. These will hold the LINQ methods for later execution. Again, when we add the variables to the listbox the methods **Min** and **Max** get executed.

Run your programme again. When you click your button, you should see this:



Average

You can also get the average of all the numbers in your array. The LINQ method **Average** is used for this, not surprisingly. Here's some new code to try out:

```
double averageValue = numberList.Average();
listBox1.Items.Add("Average of all values is: " + averageValue);
```

Notice that type is now **double**. This is because the Average method returns a variable of this type.

When the code is run, LINQ will add up all the numbers in your array and divide by the number of elements, thus giving you the average.

The answer you should get when you run your programme is 3.

Contains

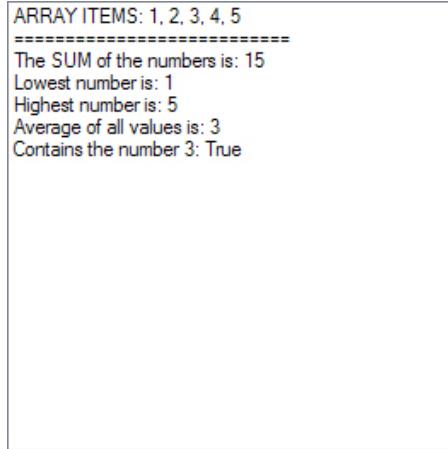
You can check to see if your array holds a certain value, the number 3 for example. For this, use the **Contains** LINQ method. Like this:

```
bool doesContain = numberList.Contains(3);
listBox1.Items.Add("Contains the number 3: " + doesContain);
```

The variable **doesContain** is a **bool**. It's **bool** because the **Contains** method returns a Boolean value.

In between the round brackets of **Contains** you type the value you want check. In the code above, we're checking to see if the array contains the number 3. If it does, a value of **True** will be returned. If it doesn't, a value of **False** will be returned.

Run your programme and test it out. With your new additions, your listbox should look like this:



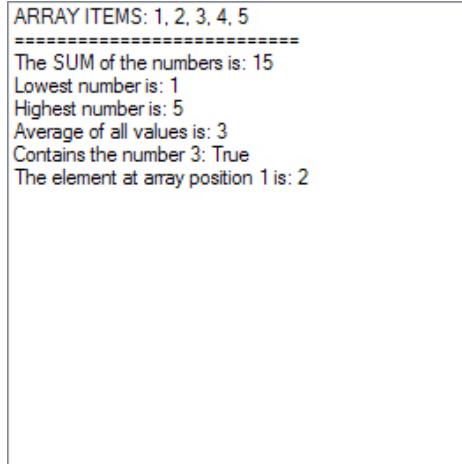
Element At

As well as checking an array to see if it contains a particular element, you can get at the element in a specified position. You do this with **ElementAt**. For example, if you want to know which number is at position 1 in your array, you can do this:

```
int elementValue = numberList.ElementAt(1);
listBox1.Items.Add("The element at array position 1 is: " + elementValue);
```

In between the round brackets of **ElementAt** you type the array position you're trying to get at.

When you run your code, your listbox will then look like this:



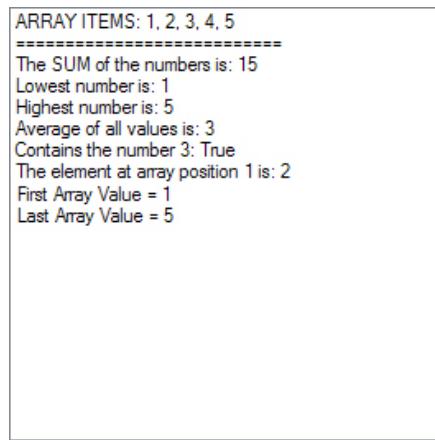
First and Last

If you need check the first element or last element in your array then you can use the LINQ methods **First** and **Last**. Try out the following:

```
int firstElement = numberList.First();
listBox1.Items.Add("First Array Value = " + firstElement);

int lastElement = numberList.Last();
listBox1.Items.Add("Last Array Value = " + lastElement);
```

When you run your programme and click your button, your listbox will look like this:



Distinct

You can check an array to see if it contains duplicates, and return only the non-duplicate values. For this, the **Distinct** LINQ method is used. Let's see a coding example.

First, add a few lines for a heading in the listbox:

```
listBox1.Items.Add("=====");  
listBox1.Items.Add("ARRAY ITEMS: 1, 1, 2, 2, 3, 4, 5, 5");  
listBox1.Items.Add("REMOVE DUPLICATES FROM ARRAY");  
listBox1.Items.Add("=====");
```

Now set up a new array:

```
int[] aryNums = new int[8] { 1, 1, 2, 2, 3, 4, 5, 5 };
```

As you can see, the new array contains duplicates – the numbers 1, 2 and 5 are repeated. To get rid of them, add the following line:

```
var distinctNums = aryNums.Distinct();
```

Notice the keyword **var** at the start. You can use this instead of a type like **int**, **string**, or **bool**. It's useful when you're not sure what type of value or values will be returned. The keyword **var** is not a variable type itself. C# will infer the type of variable it is when the programme is run.

By using **Distinct** after your array name, LINQ will return just the non-duplicates. To execute the method, add the follow **For Each** loop:

```
foreach (var num in distinctNums)  
{  
    listBox1.Items.Add(num.ToString());  
}
```

The variable **distinctNums** will contain only the non-duplicates numbers. The **foreach** loop goes round each one, adding it to the listbox.

When you run your programme and click your button, the listbox should look like this:

```

ARRAY ITEMS: 1, 2, 3, 4, 5
=====
The SUM of the numbers is: 15
Lowest number is: 1
Highest number is: 5
Average of all values is: 3
Contains the number 3: True
The element at array position 1 is: 2
First Array Value = 1
Last Array Value = 5
=====
ARRAY ITEMS: 1, 1, 2, 2, 3, 4, 5, 5
REMOVE DUPLICATES FROM ARRAY
=====
1
2
3
4
5

```

So just by using the **Distinct** LINQ method, we've quickly removed all the duplicates from an array.

Take and Intersect

Two more useful LINQ methods are **Take** and **Intersect**. As an example of how to use these, what we'll do is set up an array of winning lottery numbers. We can then take the first 6 and check them against a player's chosen lottery numbers

So add a new button to your form. Double click the new button and add the following two arrays:

```

int[] lotNums = new int[10] { 43, 31, 7, 22, 29, 16, 10, 4, 7, 41 };
int[] chosen = new int[6] { 31, 9, 8, 43, 22, 1 };

```

If you were coding a real lottery programme, you'd set up an array with the numbers 1 to 49 (UK lottery). You'd then use an algorithm to jumble the numbers up. That would leave you with an array of random numbers, 1 to 49. If you then peeled off the first 6 numbers, you could use those for your winning numbers. For the **lotNums** array above, we've just hard-coded the first 10 numbers to simulate our lottery, rather than type out all 49 of them. The second array above represents a player's chosen lottery numbers.

The first task, then, is to peel off the first 6 numbers from the **lotNums** **array**. These will represent the winning numbers of our lottery. The LINQ method that can do this quite easily is **Take**. Add the following line to your code:

```
var winners = lotNums.Take(6);
```

In between the round brackets of the **Take** method you type a number. The number is how many elements you want to take from your array, starting at the first element.

To execute the above method, add the following **foreach** loop:

```
listBox1.Items.Add("=====");  
  
foreach (var num in winners)  
{  
    listBox1.Items.Add(num.ToString());  
}  
  
listBox1.Items.Add("=====");
```

When you run your programme and click your new button, you should find that the first 6 numbers from your **lotNums** array appear in the listbox.

Stop your programme and return to your code.

You can check one array against another to see how many numbers from array A appear in array B. This is done with the **Intersect** method. Add the following lines to your code:

```
listBox1.Items.Add("=====");  
  
var myNumbers = chosen.Intersect(winners);
```

The first array goes to the right of the equal sign. For us, this is the **chosen** array, the one that contains the player's numbers. After a dot, you then type **Intersect**. Between the round brackets of **Intersect** you type the second array. LINQ will then check which numbers are included in both arrays.

Add another **foreach** loop to test things out:

```
foreach (var numbers in myNumbers)  
{  
    listBox1.Items.Add(numbers.ToString());  
}  
  
listBox1.Items.Add("=====");
```

We can also use the **Count** method to see how many elements are in **myNumbers**. Add the following line:

```
listBox1.Items.Add("Number of winners:" + myNumbers.Count() );
```

Run your programme and click your button. Your list box will look like this:

```
43
31
7
22
29
16
=====
31
43
22
=====
Number of winners:3
```

First, we have the six numbers we peeled off with **Take**. Next, we have the three numbers we got with **Intersect**. And finally, how many numbers were in both arrays, which we got with **Count**.

So we covered quite a few useful LINQ methods. But LINQ is a lot more than just a set of inbuilt methods. We'll now look at ways we can query data with keywords such as FROM, IN, WHERE, SELECT, and ORDER BY.

LINQ Queries

If you want to query your data and select only certain items, depending on a set of criteria, then LINQ queries are what you need. A LINQ query consists of words like FROM and IN and WHERE which you use to filter your data. Let's make a start.

You can use your project from the previous section for this. Add a new button to your form. Double click your new button to get at the code. Now add the following string array:

```
string[] countries = new string[17] {  
    "Antarctica", "Akrotiri", "Algeria", "Albania", "Australia",  
    "Andorra", "Angola", "Anguilla", "Antigua and Barbuda",  
    "Ashmore and Cartier Islands", "Argentina", "Azerbaijan",  
    "Armenia", "American Samoa", "Afghanistan", "Austria", "Aruba"};
```

So we're just setting up an array of countries. They are not sorted alphabetically, as we can do this with a LINQ query.

FROM, IN, SELECT

Next, add the following rather curious code: (Notice the keywords we've highlighted in red)

```
var query = from country in countries select country;
```

A LINQ query goes to the right of an equal sign. To make it easier to read, you can spread it out over more than one line:

```
Dim query = from country in countries  
    select country
```

The first part says **from country in countries**. What we want to do here is to get at each individual item in our array. To do that, we've set up a new variable and called it **country**. But we could have called this new variable almost anything we liked – it's just a variable name. So all we're saying here is "From the countries array access an individual country".

The final keyword is **select**. This is used to do the actual selecting from the array.

To the left of the equal sign is another variable, which we've called **query**. Like LINQ methods, LINQ queries aren't executed straightaway. So the variable called **query** doesn't hold the results – it just stores the LINQ until it's needed.

To execute the query we can use a **foreach** loop again. Add the following to your code:

```
foreach (var value in query)
{
    listBox1.Items.Add(value);
}
```

The whole of your code should look like this:

```
private void button2_Click(object sender, EventArgs e)
{
    string[] countries = new string[17] {
        "Antarctica", "Akrotiri", "Algeria", "Albania", "Australia",
        "Andorra", "Angola", "Anguilla", "Antigua and Barbuda",
        "Ashmore and Cartier Islands", "Argentina", "Azerbaijan",
        "Armenia", "American Samoa", "Afghanistan", "Austria", "Aruba"};
    var query = from country in countries select country;
    foreach (var value in query)
    {
        listBox1.Items.Add(value);
    }
}
```

We've set up another variable in the **foreach** line and called it **value**. The **query** variable, remember, is holding the query definition. We're transferring each item that the query returns into this new **value** variable. We're then displaying it in a listbox.

Run your code and try it out. When you click your button, your listbox should fill up:

Antarctica
Akrotiri
Algeria
Albania
Australia
Andorra
Angola
Anguilla
Antigua and Barbuda
Ashmore and Cartier Islands
Argentina
Azerbaijan
Armenia
American Samoa
Afghanistan
Austria
Aruba

However, the above code is not very useful because we could have gotten the same result by saying **foreach value in countries**. What will make it useful is we could filter the array.

Order By

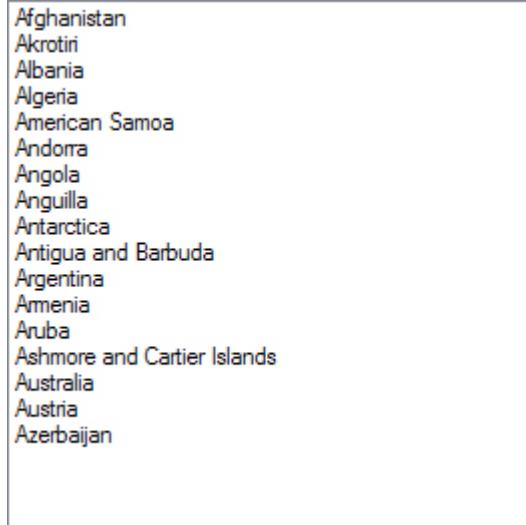
You can sort your arrays or collections quite easily by using the keywords **orderby**. After **orderby** you then add the column you want to use as a sort. We only have one column (one list of items), but if this were a query on a database, we'd have more columns we could choose from as a sort.

To see what **orderby** does, amend your LINQ to this:

```
var query = from country in countries
            orderby country
            select country;
```

So we want to sort by the country column, which is our only one.

Run your programme again, once you've added the **orderby** line. This time, your listbox will look like this when you click your button:



As you can see, the list of countries is now sorted alphabetically in ascending order, which is the default. If you want a descending sort then add the word **Descending**:

orderby country descending

You can also do multiple sorts. For example, suppose you want an alphabetical sort and a sort on the length of the country names. You could do it like this:

orderby country.Length, country

Note that a comma separates the two sorts. When you run the programme the listbox will look like this:

```
Aruba
Angola
Albania
Algeria
Andorra
Amenia
Austria
Akrotiri
Anguilla
Argentina
Australia
Antarctica
Azerbaijan
Afghanistan
American Samoa
Antigua and Barbuda
Ashmore and Cartier Islands
```

WHERE

You can refine and filter your query with the WHERE keyword. Suppose we wanted to select only countries that have 10 characters or more. You can do so by using the **Length** property. Like this:

where country.Length > 10

You've met the **>** operator before – it means greater than. But you can use any of the other logical operators that you're familiar with. If you wanted to return the countries that have less than 10 characters, for example, you'd do this:

where country.Length < 10

And if you wanted the countries that have between 7 and 10 characters, you could do this:

where country.Length > 7 && country.Length < 10

This time, we've added the **&&** (AND) logical operator between the two Lengths.

To test it out, add the WHERE part to your code:

```
var query = from country in countries
            where country.Length > 10
            orderby country
            select country;
```

When you run your programme, you should find that the listbox only contains four values.

Exercise

Try the following WHERE clause:

```
where country.Length < 7
```

How many countries did the query return?

Now try this one:

```
where country.Length >= 7 && country.Length <= 10
```

How many countries did this new query return?

Some of the other LINQ methods are available when you use the WHERE keyword. You can filter your queries by using **StartsWith** and **EndsWith**, for example. Suppose we wanted to return only the countries that began with the letters "An". We could do it like this:

```
var query = from country in countries
            where country.StartsWith("An")
            orderby country
            select country;
```

In between the round brackets of **StartsWith**, we've typed the letters we want to filter for. Note that the search is case sensitive, so "an" is a different search from "An".

If, however, we wanted a filter that selected only those countries that ended in "ia" we could do it like this:

```
var query = from country in countries
            where country.EndsWith("ia")
            orderby country
            select country;
```

Try both of the queries above out and see how many results you get back each time.

We'll leave LINQ there. It is a vast subject, though, and well worth exploring further.

Events

In programming terms, an Event is when something special happens. Inbuilt code gets activated when the event happens. The event is then said to be “Handled”. The Events that we’ll discuss in this section are called GUI Events (GUI stand for Graphic User Interface). They are things like clicking a mouse button, leaving a text box, right clicking, and many more. We’ll start with the Click event of buttons.

The Click Event for Buttons

The click event gets activated when a button is clicked on. Examine the default code for a button:

```
private void button1_Click(object sender, EventArgs e)
{
}
```

In between the round brackets, we have this:

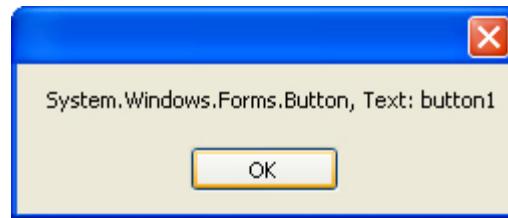
object sender, EventArgs e

The **object** keyword refers to the object which activated the event, a button in this case. This is being placed in a variable called **sender**. You can test this for yourself.

Start a new project. Add a button to your new form and double click it. Place the following code between the curly brackets:

```
MessageBox.Show( sender.ToString() );
```

We’re just using the **ToString** method on the **sender** variable. Run your programme and click the button. You should see this:



The Message is displaying which object was the sender of the event, as well as displaying the Text property of the sender: the button with the Text “button1”.

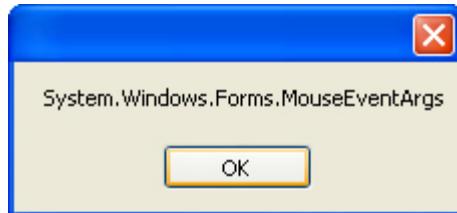
The other argument in between the round brackets was this:

EventArgs e

EventArgs is a class. It's short for event arguments, and tells you which event was raised. The letter "e" sets up a variable to use this class. If you change your line of code to this:

```
MessageBox.Show( e.ToString() );
```

The message box will then display the following:

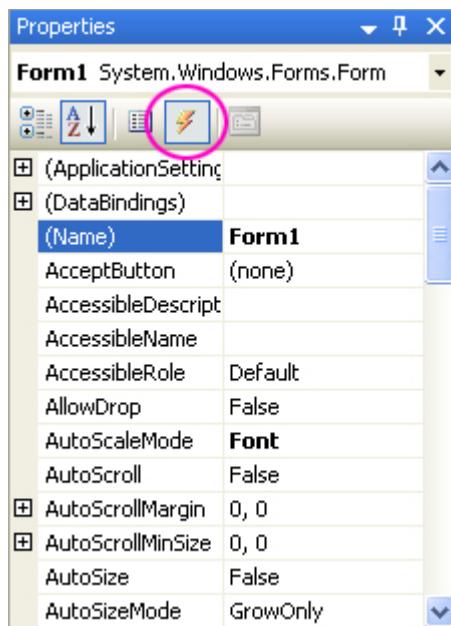


So clicking raises a Mouse Event Argument. C# .NET already knows what to do with an event of this kind, so you don't need to write any special code yourself. But what if you wanted to know which button was clicked, the left button or the right?

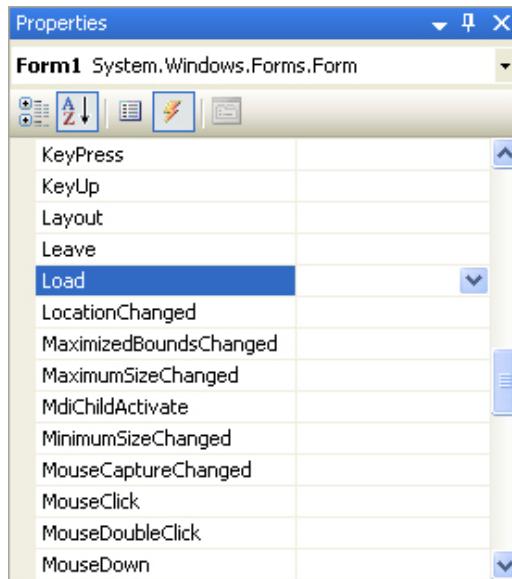
The MouseDown Event

You can see what events are available for a particular control in the Properties area.

In Design View, click on your Form1 to select it, instead of the button. To see what events are available for the Form itself, click the lightning bolt at the top of the Properties area, as in the image below:



When you click the lightning bolt, you'll see a list of events appear:



You've already met the Load event, so we won't cover it here. But notice how many events there are.

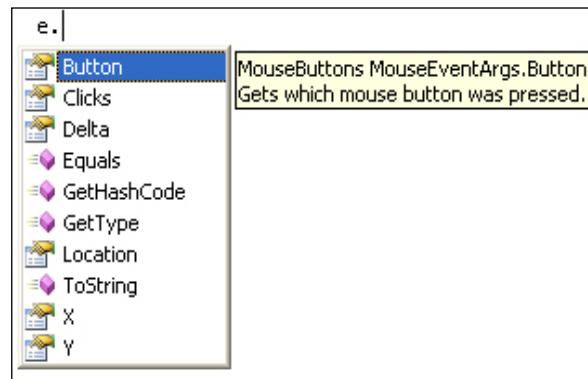
Locate the MouseDown event on the list. Now double click the word "MouseDown". You should see a code stub appear:

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
    |
}
```

In between the round brackets, there is still a **sender** object. But notice the new argument:

MouseEventArgs e

The letter "e" is the default variable name. The type of variable belongs to the MouseEventArgs. You can see what this does by typing the letter "e", then a full stop (period). You should see the IntelliSense list appear:



The list is displaying the properties and methods available to the **e** variable. One of these is the Button property.

We can use an if statement to check which mouse button was clicked. Add this to your code:

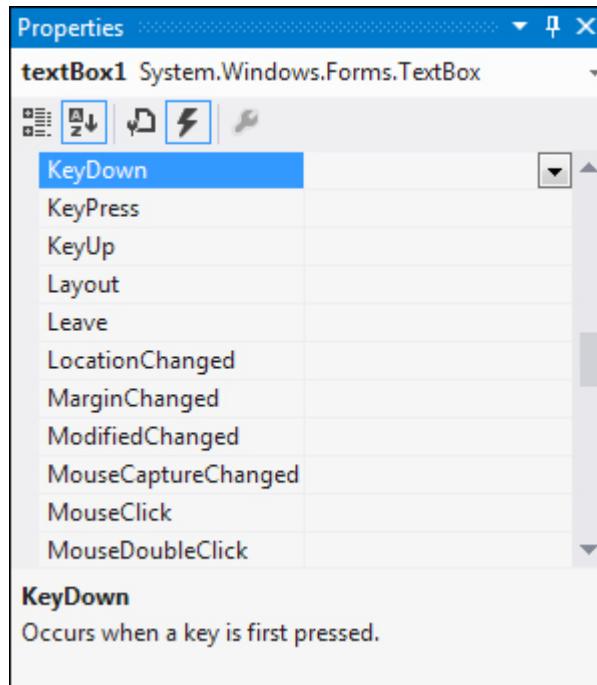
```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Left)
    {
        MessageBox.Show("Left button clicked");
    }
    else if (e.Button == MouseButtons.Right)
    {
        MessageBox.Show("Right button clicked");
    }
}
```

Run your programme and click either of your mouse buttons on the form. You should see a message box display.

The KeyDown Event

The KeyDown event works in a similar way – access an EventArgs and write code to detect which key was pressed. But a KeyDown event for a form doesn't make much sense. So add a textbox to your form.

From the Properties area on the right, locate the KeyDown events for your textbox:



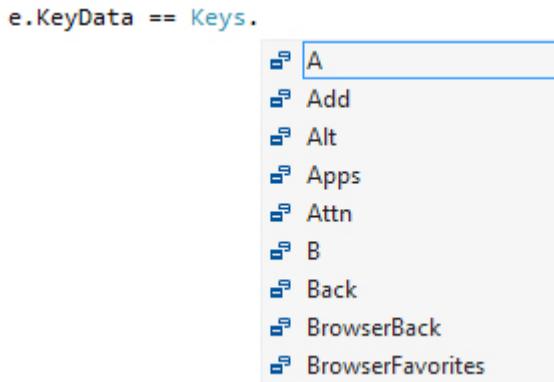
Now double click the name of the event to open up its code stub. You should see this:

```
private void textBox1_KeyDown(object sender, KeyEventArgs e)
{
}
```

This time, instead of MouseEventArg we have a **KeyEventArgs**. After typing a letter “e”, you’ll see the IntelliSense list display a list of available options. Select **KeyData** from the list:



The one you use to get at a particular key is the **KeyData** property. After a double equals sign, you then choose **Keys** from the IntelliSense list. Type another full stop and you’ll see this list:



The list contains all the letters, characters and keys from your keyboard.

Wrap all this up in an IF statement and you’ll have this:

```
if (e.KeyData == Keys.A)
{
    MessageBox.Show("Letter A Pressed");
}
```

Run your form and test it out. Click inside your textbox. You should see the message box appear when you press the letter “a” on your keyboard.

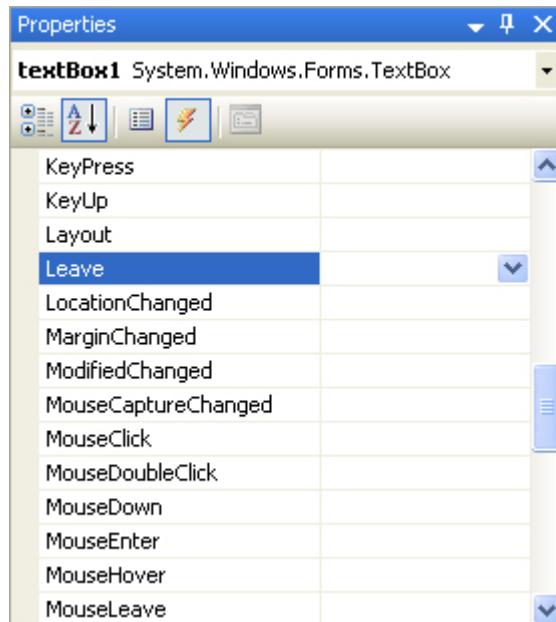
The Leave Event

A very useful event you can use for text boxes is the Leave event. It allows you to validate a text box when a user tries to leave it. You can check, for example, if the text box is blank. If it needs to be filled in, you can keep them there. Let’s try that.

Add another text box to your form. Locate the **TabIndex** property for textbox1 and set it to 1. For textbox2, set the **TabIndex** property to 2. (The button you added will be TabIndex 0.)

The **TabIndex** property refers to which control will be selected when the Tab key is pressed. On a form that a user has to fill in, you want the cursor to jump to the next textbox. You don't want the cursor jumping down from textbox1 to textbox8 – it has to be a nice sequential order. The **TabIndex** allows you to set the Tab order for all the controls on your form.

With TextBox1 selected, click the lightning bolt in the properties area to see a list of events for textboxes. You should see this:



Double click the **Leave** event to bring up its code stub. Now enter the following code:

```
private void textBox1_Leave(object sender, EventArgs e)
{
    if (textBox1.Text == "")
    {
        MessageBox.Show("You can't leave this box blank");

        textBox1.Focus();
    }
}
```

The code just checks for a blank text box. But it makes the check when the user attempts to leave the text box and move on to the next one. Notice how the user is brought back to the text box:

textBox1.Focus();

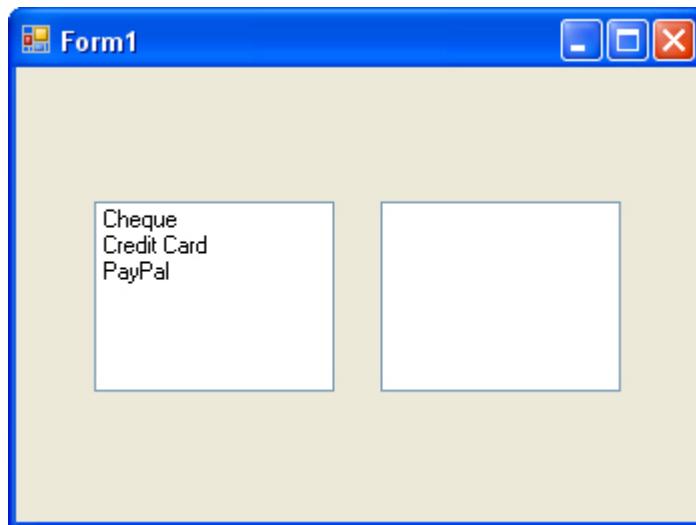
The Focus method can be used to force a control to be selected. For text boxes, this means that the cursor will be flashing inside of it.

You can also do things like converting text to upper or proper case when the users Tabs away from a text box. Here's some code that converts to uppercase:

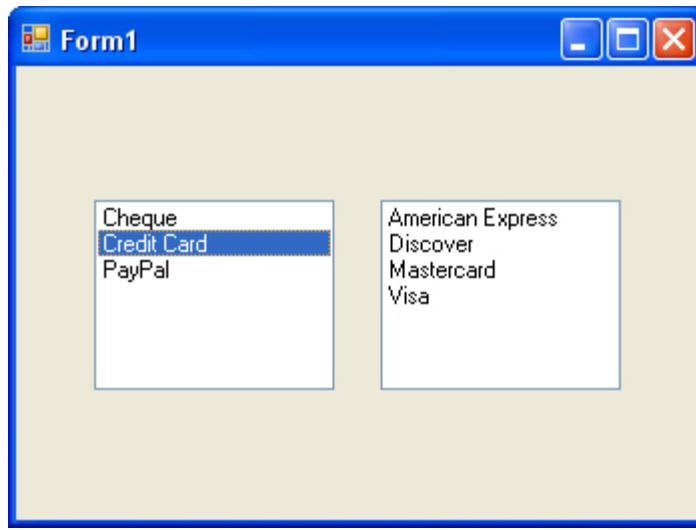
```
private void textBox1_Leave(object sender, EventArgs e)
{
    if (textBox1.Text == "")
    {
        MessageBox.Show("You can't leave this box blank");
        textBox1.Focus();
    }
    else
    {
        textBox1.Text = textBox1.Text.ToUpper();
    }
}
```

ListBox and ComboBox Events

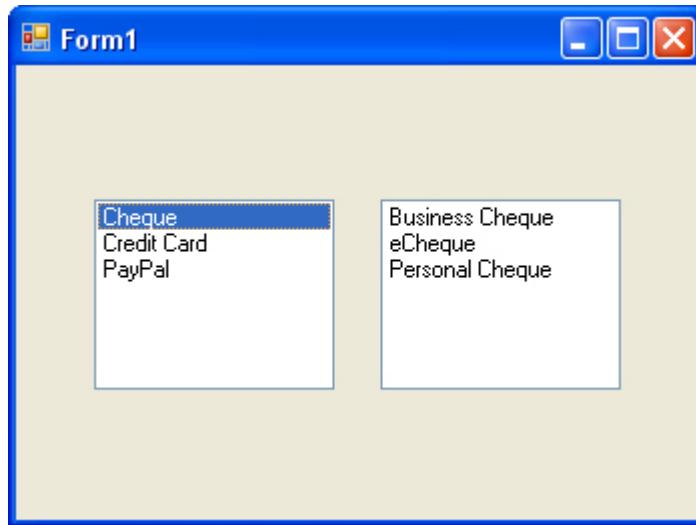
A useful event you may want to use for a ListBox or a ComboBox is the SelectedIndexChanged event. You can use this event to get which item in your List or Combo box was selected. The event will fire again when you select a new item. Why is this useful? Well, as an example, you can populate a second ListBox or ComboBox based on which item was selected in the first one. Examine the image below:



We have two ListBoxes on a form. The first one shows payment methods, and the second one is blank. When Credit Card is clicked in the first ListBox, notice what happens to the second ListBox:

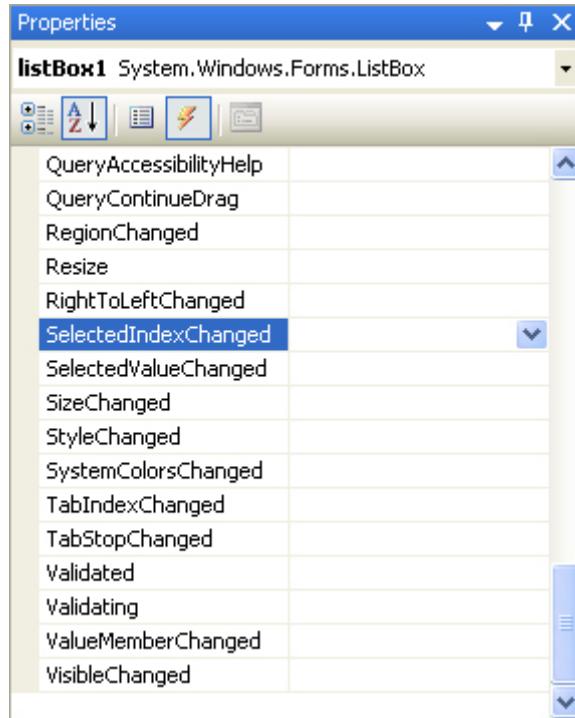


The second ListBox is now displaying the types of credit cards accepted. Click on the Cheque item, and different options are available:



All this happens with the SelectedIndexChanged event.

To see how the code works, start a new project and put two ListBoxes on your form. Click on the first one to select it, and use the **Item** property to add the three items: Cheque, Credit Card, and PayPal. Now click on the lightning bolt to display the event for a listbox:



Double click the SelectedIndexChanged event to bring up its code stub:

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    |
}
```

To get at which item in a ListBox or ComboBox was selected, you use the SelectedIndex property. The first item in the list is item 0, the second item 1, etc. So add this if statement to your code:

```
if (listBox1.SelectedIndex == 1)
{
    loadListBox();
}
```

So we're just checking if the selected index has a value of 1. If it does, we're calling a method – **loadListBox**. Add the method to your code:

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    if (listBox1.SelectedIndex == 1)
    {
        loadListBox();
    }
}

private void loadListBox()

{
    listBox2.Items.Clear();

    listBox2.Items.Add("American Express");
    listBox2.Items.Add("Discover");
    listBox2.Items.Add("Mastercard");
    listBox2.Items.Add("Visa");
}
```

You've already covered ListBoxes in an earlier section, so we won't explain how it works. But Run your programme and test it out. You should find that it works as described above.

Exercise N

Add a second method that loads items when the Cheque payment method is selected. Call the method from the SelectedIndexChanged event when item 0 is selected in the first ListBox. When you programme runs, the Cheque payment options should display in your second ListBox, instead of the Credit Card options.

[Answer to Exercise N](#)

Classes and Objects in C# .NET

C# .NET is an Object Oriented programming language. Objects are created from Classes, which you've already used a lot. The Windows Form itself is a Class, and when you run your programme, you are creating Objects: a Form object, a button object, a Textbox object, etc. In this section, you'll learn how to write your own Classes, and turn them in to Objects. First, an explanation

What is a Class?

A Class is simply a chunk of code that does a particular job. You might have a class that handles all your database work, for example, or one that does error checking on Textboxes.

The idea is that you can reuse this code (Class) whenever you need it, or when you need it in other projects. It saves you from having to write the same thing over and over again.

Think of a Class as a recipe. If you had a recipe for a delicious Banana Cake, the recipe will tell you what you need to do to make the cake. But it's not the cake itself. It's the instructions for the cake. If you have the recipe, you can make Banana Cakes whenever you need them.

What is an Object?

An Object is the thing that the recipe makes - The Banana Cake itself. You do all the coding in a Class (recipe), and then instruct C# to make an object (a Banana Cake). The two processes are different.

But all you are really doing with Classes and Objects is trying to separate code into chunks, so that it can be reused. When the code is being used, it's an object. But the object is created from your code (Class).

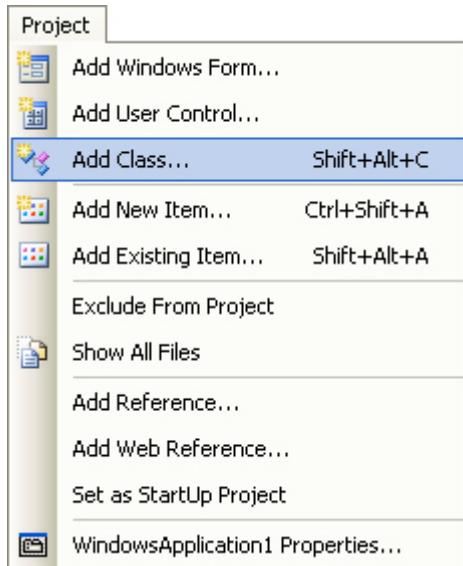
Let's get some practice.

A Happy Birthday Class

The first Class that you'll create is a very simple one, and won't be that much use in the real world. But it will show you how Classes are structured, and how

Objects are created from Classes. All our Class will do is display a "Happy Birthday" Message.

So start a new project, and add a button to your form. We'll use the button later. To add a new class to your programme, click the **Project** menu at the top of C#. From the Project menu, select **Add Class**:



You should then see the **Add New Item** dialogue box popping up. Select the Class item from the available templates. In the Name box at the bottom, type this:

HappyBirthday.cs

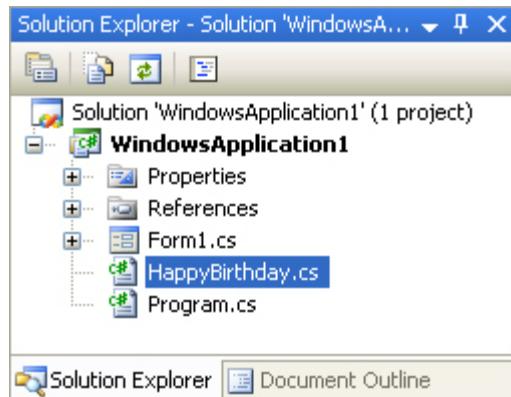
Then click the Add button on the dialogue box. You should see all the default code for your class appear (users of later versions of Visual Studio may see more **using** statements):

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace WindowsApplication1
6  {
7      class HappyBirthday
8      {
9      }
10 }
```

WindowsApplication1 was the default name when we created a new project. If we had changed the default project name to, say, Birthday then it would have said **namespace Birthday**. (A namespace is where classes are grouped together. System is a namespace. As too are Collections, Generic, and Text.)

Notice that the Solution Explorer on the right is showing your Class:



The class called **HappyBirthday.cs** is now an item under Form1.cs. The **.cs** extension means that the file is a Class.

The Class code stub that C# created for us, though, was this:

```
class HappyBirthday
{
}
```

So you start with the word “class” followed by a space. The name of your class comes next, followed by a pair of curly brackets. All the code for your class goes between the two curly brackets.

Let’s add a simple message to the class.

What we’ll do is to create a Method in the class that displays a message. So add this to your code:

```
class HappyBirthday
{
    public string getMessage()
    {
        return "Happy Birthday";
    }
}
```

The name of our Method is **getMessage**, but we could have called it almost anything we liked. The Method is going to return a **string**. Notice that we’ve made the Method **public**. A public Method (or variable) set up inside of a class means that it can be seen from the outside. So a button on a form can see it, for example. If you make it **private** then it can only be seen from inside of the class.

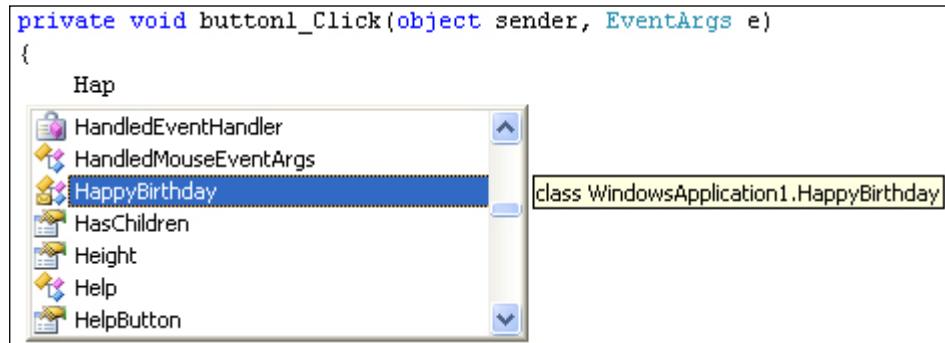
This is just a normal Method, though, and you’ve covered these in an earlier section. The only difference this time is that it’s in a Class of its own.

Creating Objects from your Classes

Now that we have added a Method to our Class, let's turn it into an Object.

Go back to your form, and double click the button you added. This will bring up the code stub for the button.

To create an object from a class, you have to set up a variable of that class type. This involves nothing more than typing its name. So type the “Hap” of HappyBirthday (the name we gave our class). You should see the IntelliSense list appear with you Class on it:



Press the enter or tab key to add the Class name to your code.

After the Class type, you need a name for your variable. This is just like any other variable name, so you can call it almost anything you like. But type **birthdayMessage** as the name of your **HappyBirthday** class variable, and your code will look like this:

```
private void button1_Click(object sender, EventArgs e)
{
    HappyBirthday birthdayMessage;
}
```

So this sets up a variable of type HappyBirthday (your class), with the name **birthdayMessage**.

We haven't yet created the Object, though. All we've done is to tell C# to set up some space in memory to hold the Object.

You create Objects with the **new** keyword. Like this:

variable_name = new class_name();

So the name of your variable goes on the left of an equals sign. To the right of the equals sign, you need the **new** keyword. Type a space and the name of your Class again. This time, you need a pair of round brackets after the class name. End the line with a semicolon, as normal.

Add the following line to your code:

```
birthdayMessage = new HappyBirthday();
```

Your code window should then look like this:

```
private void button1_Click(object sender, EventArgs e)
{
    HappyBirthday birthdayMessage;
    birthdayMessage = new HappyBirthday();
}
```

If you prefer, you can keep everything on one line. Like this:

```
HappyBirthday birthdayMessage = new HappyBirthday();
```

But bear in mind that you are doing two things at once here: Setting up the variable in memory, and then creating a new Object.

Now that we've created an object, we can go ahead and call the one method that is in our Class.

Now that you have an Object, you can use the dot notation to call your Method:

```
Object_Name.Method_Name()
```

Inside of a MessageBox, type the name of your Object. Then type a full stop. You should then see the IntelliSense list appear with your Method on it:

The screenshot shows a code editor with the following line of code:

```
MessageBox.Show( birthdayMessage. );
```

An IntelliSense dropdown menu is open at the end of the line, listing several methods:

- Equals
- GetHashCode
- getMessage** (highlighted)
- GetType
- ToString

To the right of the dropdown, a tooltip displays the signature of the selected method:

```
string HappyBirthday.getMessage()
```

The reason why it's there is because you made your Method public. If you had made it private then the Method wouldn't show up on the list from outside the class.

Add the Method to your code and you should have this:

```

private void button1_Click(object sender, EventArgs e)
{
    HappyBirthday birthdayMessage;
    birthdayMessage = new HappyBirthday();

    MessageBox.Show( birthdayMessage.getMessage() );
}

```

Run your programme, and click the button on your form. You should see a message appear.

Congratulations – you've created your very first Class!

Passing values to your classes

The message that displays is just “Happy Birthday”. But what if you wanted the name of a person as well, “Happy Birthday, Shahid!”? What you need is a way to pass information over to your classes.

There are several ways to pass data to your classes. An easy way is to simply pass the information as an argument, between the round brackets of the method.

Change the Method in your class to this (the new additions are in bold):

```

public string getMessage(string givenName)
{
    return "Happy Birthday " + givenName;
}

```

So we've just added a string variable called **givenName** between the round brackets of the Method. We then use this in the code.

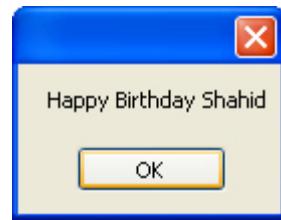
If you try to start your programme, however, you'll get an error message. C# will underline this part of your code:

birthdayMessage.getMessage()

Because you've added an argument to your method, you need some data between the round brackets. Change the line to this:

birthdayMessage.getMessage(“Shahid”)

Run your programme again, and you should find that the following message displays:



So you're just passing data to your Method in the normal way.

However, we set the method up to be public:

```
public string getMessage(string givenName)
```

This is not the recommended way to pass information over to your classes, though. Methods are usually hidden from the outside world by making them **private**. This “hiding” methods and variables in your class is known as **Encapsulation**. The idea is that you hide as much of your class as possible. That way, it can't be broken by passing information over that the class can't handle. A broken class may crash your programme!

So change the first line of your Method to this:

```
private string getMessage(string givenName)
```

Try to run your programme and you'll get an error:

“getMessage is inaccessible due to its protection level.”

The reason it's inaccessible is because you've made it private. So it can't be seen from outside of its own class. If that's the case, then how can you pass values over? The answer is to set up a property instead.

Adding Properties to your Class

You've been using properties a lot throughout this book: setting the Text property for a Textbox, setting size properties, setting font properties, etc. The way you did it in code was this:

```
textBox1.Text = "My Text";
```

So the object is called **textBox1**. This **textBox1** object has a property called **Text**. To set a value for the **Text** property, you're typing a string between double quotes, and after an equals sign (=).

Behind the scenes, this property value gets handed over to a class, so that something can be done with it.

C# .NET allows you to set up your own properties, so that they can be used the same way. Let's see how it's done.

The only thing you have in your class at the moment is a private method:

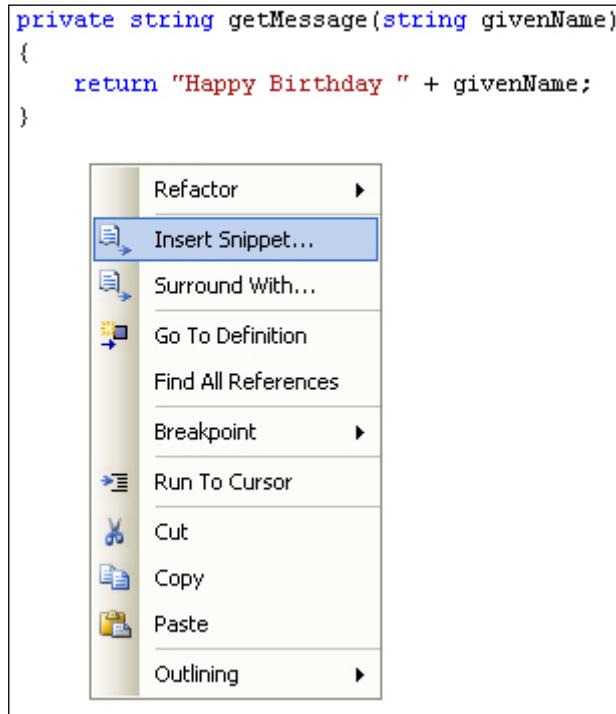
```
private string getMessage(string givenName)
{
    return "Happy Birthday " + givenName;
}
```

This Method can't be seen from the outside world. But that doesn't mean that it can't be used inside of the class. We'll have our property call this method.

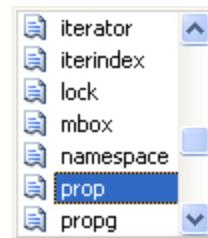
But our property will do the following:

- Allow us to pass a value over to our class
- Allow us to get that value back out of the class

The easy way to add a property to your class is via the **Insert Snippet** menu. So right click just after the final curly bracket of your **getMessage** Method. You should see the following menu appear:



Select **Insert Snippet** to see the following (Visual Studio 2012 users will see another menu first, with C# on it. Double click C# to see the menu below):



Select **prop** from the list and C# will add this default code (Visual Studio 2012 users should select **propfull** from the menu instead of prop):

```
private int myVar;

public int MyProperty
{
    get { return myVar; }
    set { myVar = value; }
}
```

The code is a little bit baffling, so we'll go through what's been added.

The default code is for an integer property called **MyProperty**. A private integer variable has also been set up. (Variables inside of your own classes are called **Fields**.)

But we want to return a birthday message as a string, not as an integer. So change your property to this:

```
private string birthdayMessage;

public string MyProperty
{
    get { return birthdayMessage; }
    set { birthdayMessage = value; }
}
```

So the name of the variable outside of the property has been changed to **birthdayMessage**:

```
private string birthdayMessage;
```

This means that the variable will now be visible from everywhere in the class.

Note that we've also replaced myVar with **birthdayMessage** inside of the property.

The structure of the property is this:

```
public string MyProperty
{
}
```

This sets a property called **MyProperty**. You can change this to almost anything you like – it's just the same as a variable name.

Inside of the property body, you have an area that **gets** values out:

```
get { return birthdayMessage; }
```

And an area where you can put values in (**set** them):

```
set { birthdayMessage = value; }
```

The keywords **get** and **set** are used for this purpose. The **value** is the value you will be passing in to the property. Change the line to this:

```
birthdayMessage = getMessage( value );
```

We are now using our private `getMessage()` Method. The value being handed over to our Method will be the **value** that gets handed over to the property.

(If you only need to hand values over and not get them out, then just use the **set** part. This is known as a write-only property. If you just want to get values out, then just use the **get** part. The property will then be read-only.)

Using your Properties

Now that you've set up a property in your class, you can put it to use. First, comment out any code for your first button. Place a new button on your form. Double click the button to get at the code.

The first thing to do is to create an object from your class:

```
private void button2_Click(object sender, EventArgs e)
{
    HappyBirthday birthdayMessage = new HappyBirthday();
}
```

In the above code, we've created a `HappyBirthday` object and called it **birthdayMessage**.

To pass something over to your new property, you use dot notation. So type **birthdayMessage** followed by a dot (full stop). You should see the IntelliSense list appear:



And there's the property you set up! The name of the property is **MyProperty**. (That, however, is the default name. You can call your properties almost anything you like, just as you can with variables.)

Double click your property to add it to your code.

To pass something in, you need an equals sign, followed by the information you want to pass to your property:

```
birthdayMessage.MyProperty = "Shahid";
```

In the code above, we're passing the text "Shahid" to our property.

To get something back out from your property, the syntax is this:

```
your_data = object_variable_name.property_name;
```

After the equals sign, you need the name of your object. Then type a dot. The IntelliSense list should appear again. Select your property from the list. End the line with the usual semicolon.

So add the following lines to your code:

```
private void button2_Click(object sender, EventArgs e)
{
    HappyBirthday birthdayMessage = new HappyBirthday();
    string returnedMessage;

    birthdayMessage.MyProperty = "Shahid";
    returnedMessage = birthdayMessage.MyProperty;

    MessageBox.Show(returnedMessage);
}
```

Note the two lines that use the property:

```
birthdayMessage.MyProperty = "Shahid";
returnedMessage = birthdayMessage.MyProperty;
```

The first one is **setting** a value for the property. The second line is **getting** something back out. We're then handing it over to a string variable called **returnedMessage**. This can then be displayed in a message box.

If you're not sure quite what's going in, study the following images:

```
private void button2_Click(object sender, EventArgs e)
{
    HappyBirthday birthdayMessage = new HappyBirthday();
    string returnedMessage;

    birthdayMessage.MyProperty = "Shahid";
    returnedMessage = birthdayMessage.MyProperty;

    MessageBox.Show(returnedMessage);
}
```

```
class HappyBirthday
{
    private string getMessage(string givenName)
    {
        return "Happy Birthday " + givenName;
    }

    private string birthdayMessage;

    public string MyProperty
    {
        get { return birthdayMessage; }

        set { birthdayMessage = getMessage(value); }
    }
}
```

The image shows that the text “Shahid” from the button is getting handed over to the **value** variable in the **set** part of the property. The **getMessage()** method can then do something with this value. After it constructs the message, it hands it over to the **birthdayMessage** variable. The **get** part of the property can then see what's inside of the **birthdayMessage** variable.

Here's a pair of images showing what is happening when we get the value out of the property:

```

private void button2_Click(object sender, EventArgs e)
{
    HappyBirthday birthdayMessage = new HappyBirthday();
    string returnedMessage;

    birthdayMessage.MyProperty = "Shahid";
    returnedMessage = birthdayMessage.MyProperty;

    MessageBox.Show(returnedMessage);
}

```

```

class HappyBirthday
{
    private string getMessage(string givenName)
    {
        return "Happy Birthday " + givenName;
    }

    private string birthdayMessage;

    public string MyProperty
    {
        get { return birthdayMessage; }
        set { birthdayMessage = getMessage(value); }
    }
}

```

So the value in the class variable **birthdayMessage** is getting handed over to **MyProperty** in the button code. Once we **get** the value from the property, it is handed over to the **returnedMessage** variable.

Class Constructors

Quite often, you'll want to set some default values for the variables you've set up in your class. This is where the Class Constructor comes in handy.

Suppose we want to add a new property to our class, the number of presents a person receives on their birthday. We can set up a new variable in our HappyBirthday class:

```
private int numberOfPresents;
```

This integer variable (a field) has no value at the moment. We can use a Constructor to set one.

A Constructor is used to set up values for your variables. When your object is created, C# looks for this Constructor and tries to set those values. If you don't add one yourself, C# automatically adds one for you. Even if you have no variables to set!

The Constructor takes the same name as your class. For us, the default Constructor will be called **HappyBirthday**, and would look like this:

```
public HappyBirthday()
{
}
```

Note that Constructor is **public**, but that it has no return type like **int**, **string**, **float**, etc. That's because Constructors don't return values – they just get on with setting up values for your variables.

A default Constructor has nothing between the round brackets after its name. If you set up your own default Constructor, rather than letting C# do it for you, then leave the round brackets empty. (You can also do something called Constructor Overloading, but that's outside the scope of this book. But this is where you pass values over to your Constructor.)

To set a value for our **numberOfPresents** variable, we just do normal variable assignment:

```
public HappyBirthday()
{
    numberOfPresents = 0;
}
```

Now, when the object is created, C# will fill our **numberOfPresents** variable with a default value we have chosen.

Let's put the **numberOfPresents** variable to some use.

Add the following property to your class:

```
public int PresentCount
{
    set { numberOfPresents = value; }
}
```

The property is write-only, meaning we are only setting a value, not getting one out. But it will set a value for the **numberOfPresents** variable.

Your **getMessage** Method can be amended slightly, too. Change it to this:

```
private string getMessage(string givenName) {
    string theMessage;
```

```
theMessage = "Happy Birthday " + givenName + "\n";
theMessage += "Number of presents = " + numberOfPresents.ToString();

return theMessage;
}
```

All we're doing is setting up a string variable called **theMessage**. We're then building up the string with the **givenName** and the **numberOfPresents**. Note the use of the C# new line character, "\n".

In your button code, you can now set a value for your new property:

```
birthdayMessage.PresentCount = 5;
```

The coding window for your button should then look like this:

```
private void button2_Click(object sender, EventArgs e)
{
    HappyBirthday birthdayMessage = new HappyBirthday();
    string returnedMessage;

    birthdayMessage.PresentCount = 5;
    birthdayMessage.MyProperty = "Shahid";
    returnedMessage = birthdayMessage.MyProperty;

    MessageBox.Show(returnedMessage);
}
```

The new property use has been highlighted. We're passing a value of 5 over to the PresentCount property we set up.

Here's what your HappyBirthday class should look like. We've added a few comments.

```
class HappyBirthday
{
    //=====
    // CLASS VARIABLES
    //=====
    private int numberOfPresents;
    private string birthdayMessage;

    //=====
    // DEFAULT CONSTRUCTOR
    //=====
    public HappyBirthday()
    {
        numberOfPresents = 0;
    }

    //=====
    // METHOD
    //=====
    private string getMessage(string givenName)
    {
        string theMessage;

        theMessage = "Happy Birthday " + givenName + "\n";
        theMessage += "number of presents = " + numberOfPresents.ToString();

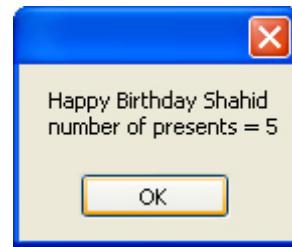
        return theMessage;
    }

    //=====
    // READ AND WRITE PROPERTY
    //=====
    public string MyProperty
    {
        get { return birthdayMessage; }

        set { birthdayMessage = getMessage(value); }
    }

    //=====
    // WRITE-ONLY PROPERTY
    //=====
    public int PresentCount
    {
        set { numberOfPresents = value; }
    }
}
```

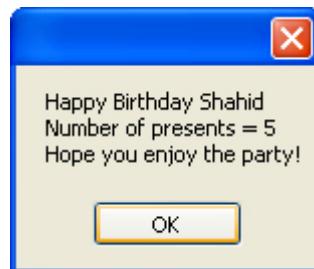
Run your programme and click your button. The message that displays should look like this:



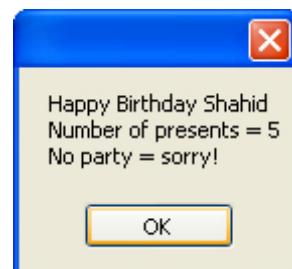
Exercise O

Set up a new property to record whether a person is having a birthday party or not. Add a default value of **false** to your constructor. Adapt your message accordingly. When you use your new property from the button code, pass over a value of **true**.

If you successfully complete this exercise, your message box may look something like this if they are having a party:



And like this if they are not:



To get different messages, you will have to add an if statement to your Method.

This is a tricky exercise. So give yourself a pat on the back, if you complete it!

[Answer to Exercise O](#)

Inheritance

Another important concept in Object Oriented programming is **Inheritance**. It's easier to show you what inheritance is through programming examples, rather than explain in words. But essentially inheritance is creating one class from another. The parent class is the main one, and you create a child class that does similar but slightly different things.

Let's see if we can't make things clearer with a programming example.

Take a look at the new version of our Happy Birthday class. It's been stripped down:

```
class HappyBirthday
{
    private string birthdayMessage;
    private string presentString;

    public HappyBirthday()
    {
        presentString = "Number of presents: ";
        birthdayMessage = "Happy Birthday ";
    }

    public string getMessage(string givenName)
    {
        return birthdayMessage + givenName;
    }

    public string getPresents(int numPresents)
    {
        presentString = presentString + numPresents.ToString();
        return presentString;
    }
}
```

We've deleted the properties, and now have just two public Methods. The default constructor is still there, though. It just sets some default text for the two variables.

Inheritance is creating a child class from this parent. The parent is known as the **base** class. The child class is known as a **derived** class. The derived class can use all of the methods from its base class. But it will have its own code. Here's a class derived from the above Happy Birthday class:

```

//=====
//      A DERIVED CLASS
//=====
class BirthdayParty : HappyBirthday
{
    //=====
    // DEFAULT CONSTRUCTOR CALLS THE base() CONSTRUCTOR
    //=====
    public BirthdayParty() :base()
    {

    }

    //=====
    //      METHOD
    //=====
    public string getParty(bool haveParty)
    {
        if (haveParty == true){
            return "Enjoy your party!";
        }
        else {
            return "Sorry - No party for you!";
        }
    }
}

```

What we want to do is to create another class called **BirthdayParty**. Because the two are related, we can use Inheritance. This means that we don't have to start all over again. We can just reuse the **HappyBirthday** class. For example, suppose we wanted to display the following message:

**Happy Birthday Shahid!
You have 8 Presents
Enjoy your Party!**

We can get the first two messages from the base class (**HappyBirthday**). But we can get the final message from the new class, **BirthdayParty**. Birthdays and parties are closely related. But we want to keep one separate from the other. We could then add more information about the party: number of guests, location, time, etc. But we'd add this to the derived class, instead of the base class.

Note the first line from the derived class:

class BirthdayParty : HappyBirthday

You start by typing the word **class**, followed by a space. You then need to come up with a name from your new class. We called ours **BirthdayParty**. Because we want to use some of the code from the base class, we have this:

: HappyBirthday

So to specify that this is a derived class, you type a colon (:). After the colon, you type the name of the base class.

The only other thing that's different is this:

```
public BirthdayParty() :base()
{
}
```

We're setting up a default constructor called **BirthdayParty**, the same name as the new class. This is where you can set up variables for this particular class. But this constructor knows nothing about the code from its parent class, **HappyBirthday**. To tell it to run the code for the parent constructor as well, we have this:

```
:base()
```

When the programme runs, C# will initialise variables for the new derived class as well as the variables in the base class.

Here's some code that uses both classes. This is from a button on a form:

```
private void button1_Click(object sender, EventArgs e)
{
    BirthdayParty partyOn = new BirthdayParty();

    MessageBox.Show( partyOn.getMessage("Shahid") );
    MessageBox.Show( partyOn.get呈s(8) );

    MessageBox.Show( partyOn.getParty( true ) );
}
```

We've created a new **BirthdayParty** object and called it **partyOn**. Then come three message boxes. The first two call the Methods from the **HappyBirthday** class. The final one calls the Method from the derived class, which is **BirthdayParty**.

Study the code and see how it works. Create a new project and try it out.

Inheritance can get very tricky, and it's difficult to know what to put in to the base class and what to put in the derived class. But it's usual to create a base class that holds general information, and a derived class that contains more information. For example, you might have an automobile class as the base class. A car class and a motorbike class could then be derived from automobile.

Method Overloading

Something else that you can do with classes is called **Method Overloading**. This is when you want a slightly different version of a Method. It's part of another Object Oriented concept called **Polymorphism**.

As an example, take the Method in our derived class. This one:

```
public string getParty(bool haveParty) {

    if (haveParty == true){
        return "Enjoy your party!";
    }
    else {
        return "Sorry - No party for you!";
    }
}
```

All this does is to return a string, depending on whether the **haveParty** variable is true or false. But what if you wanted a different version of the message, if you wanted to add the person's name as well, for example? That's where Method Overloading comes in.

The easiest way to do it is to simply copy and paste the Method you have. To Overload it, you just add more parameters. Compare this version of the **getParty** Method:

```
public string getParty(bool haveParty, string aName) {

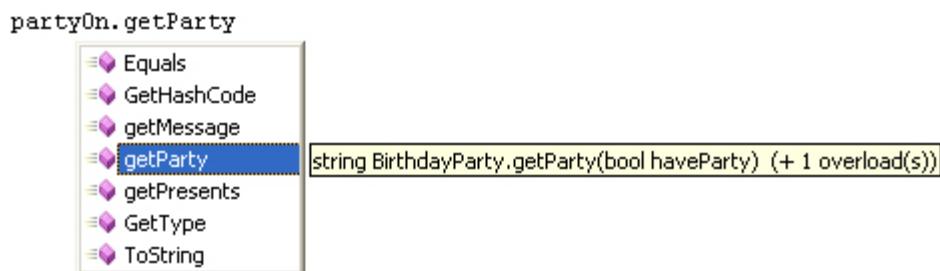
    if (haveParty == true){
        return "Enjoy your party, " + aName;
    }
    else {
        return "Sorry - No party for you, " + aName;
    }
}
```

In between the round brackets of **getParty**, we've added a second parameter:

string aName

And in the Method body, the message that gets **returned** is slightly different.

When we come to use our **getParty** Method from the button, look what happens:



The Method is showing up on the IntelliSense list. The tool tip is telling us that the Method has 1 overload.

As soon as you type the first round bracket, the Method options, the overloads, show up:

```
partyOn.getParty()
▲ 1 of 2 ▼ string BirthdayParty.getParty (bool haveParty)
```

In between the up and down arrows, it says 1 of 2. This means that there are two version of the **getParty** method. Press the down arrow to see the second one:

```
partyOn.getParty()
▲ 2 of 2 ▼ string BirthdayParty.getParty (bool haveParty, string aName)
```

The first version is our original, with one argument. The second version is the new one, with two arguments.

When you overload a method, it takes the same return type (public **string**, public **int**, public **float**, etc). It's the parameters in between the round brackets that change. In the second version of our method, the first parameter is a bool, and the second parameter is a string. But you could have this instead:

```
public string getParty( string aName )
```

So we still have one parameter, but the type of variable is a string instead of a bool. This is enough to set up a third overloaded method:

```
partyOn.getParty()
▲ 2 of 3 ▼ string BirthdayParty.getParty (string haveParty)
```

We now have 2 of 3 between the up and down arrows – our new overloaded method has been added to the list.

Overloading can come in very handy if you want slight different versions of a Method – just change the parameter list.

Before we leave classes, here's a simpler way to create classes. It's not recommend that you create all your classes like this though!

Static Methods

If all you want is an easy way to set up a few methods, and call them at will, you can set up a class filled with public static methods.

Static methods don't need an object. So you don't do this to create them:

MyClass object_name = new MyClass()

Instead, you just call them when needed. Let's see how.

From the C# menu bar at the top, click **Project > Add Class**. Call your new class **stats**. Type the following code between the curly brackets of your new class:

```
public static int addUp(int num1, int num2)
{
    return num1 + num2;
}
```

The only thing here that's different from a normal Method is the use of the word **static**. Note that it comes after **public** but before **int**. You can use any of the other variables types as well, such as **static string**, or **static bool**, etc.

Add a button to your form, and double click to get at the code.

To use the static Method in your new class, the syntax is just this:

```
class_name.method_name()
```

Because it's a static method, you don't have to go to the bother of setting up a new object – your method will available straight away.

Add the following code to you button to try it out:

```
string answer;
answer = stats.addUp(5, 4).ToString();
MessageBox.Show(answer);
```

The class name **stats** will appear on the IntelliSense menu without you having to do anything else. And when you type a dot after your class name, the static method will appear on the list.

There's an awful lot to static members, obviously. For example, you can create static fields, static properties, and even static constructors. But if you want a class filled with methods that you want quick access to, then just use static methods.

We'll leave classes now and move on. But if you want to get grips with C# .NET, especially if you want a career as a programmer, then classes are something you need to study.

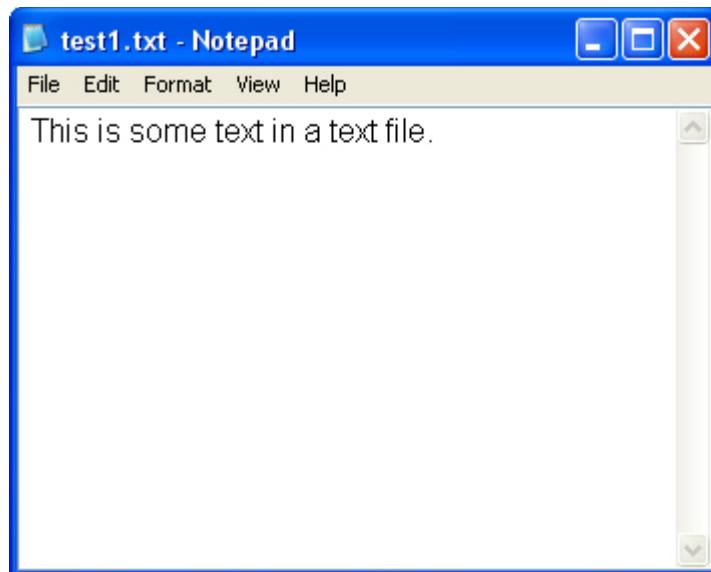
Manipulating Files with C# .NET

As a programmer, you should have the ability to manipulate files. By this we mean opening and processing things like text files, html files, word docs, etc. In this section, you'll learn how to do that. First up is opening a text file.

How to open a Text File in C#

What we'll do is to open up a plain text file and insert the contents into a text box. So start a new project. Add a text box and a button to your form. Set the **MultiLine** property of the text box to true.

Before you double click your button, you'll need to create a text file. Use the Windows Notepad to create the following text file:



(The Windows Notepad can be found by clicking **Start > All Programs > Accessories**, in XP and Vista/Windows 7. In Windows 8, just do a search for **Notepad.exe**.)

Save it anywhere you like, but we recommend saving to your root drive if you have XP. The file path would then be **C:\\test1.txt**. For Vista and Windows 7/8 users, save it to your Documents folder. The file path would then be:

"C:\\Users\\Owner\\Documents\\test1.txt"

But you can also use this rather long line:

```
string fldr = "\\test1.txt";
fldr = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) + fldr;
```

Once you have created and saved your text file, double click the button on your form to get at the code. Add your file path as a string variable. Here's the one for XP users:

```
private void btnOpen_Click(object sender, EventArgs e)
{
    string file_name = "C:\\\\test1.txt";
}
```

And here's one for Vista and Windows 7/8 users:

```
private void btnOpen_Click(object sender, EventArgs e)
{
    string file_name = "\\\\test1.txt";
    file_name = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) + file_name;
}
```

Note that we have two backslash characters (\\\\) in the file name. This is because if you just use one in C# it means “A special character follows”. The special character IS the backslash, so you need two: one to tell C# that a special character follows, and one for the special character itself. This is known as **escaping**.

To open up text files, C# .NET uses something called the **StreamReader**. This is an inbuilt class that lives in the **Input/Output** namespace. (A namespace is a collection of related classes, all grouped together.) The Input/Output namespace, or IO for short, lives in the System namespace. We need to set up a **StreamReader** variable, so the code would be this:

```
System.IO.StreamReader objReader;
```

This sets up a **StreamReader** variable that we've called **objReader**. You then create a new object with the name **objReader**:

```
objReader = new System.IO.StreamReader( file_name );
```

After the new keywords, we have **System.IO.StreamReader** again. However, after StreamReader we have a pair of round brackets. The file you want to open goes between the round brackets. For us, this was held in the variable we called **file_name**. Add the two lines to your code, and your coding window should look like ours below (this is the shorter Windows XP version):

```
private void btnOpen_Click(object sender, EventArgs e)
{
    string file_name = "C:\\\\test1.txt";

    System.IO.StreamReader objReader;
    objReader = new System.IO.StreamReader(file_name);
}
```

Now that we have a StreamReader object, we can use its properties and methods. To place the entire contents of a file in to a text box, you can use the **ReadToEnd**

method. As its name suggests, this reads all the contents of the file. Add this line to your code:

```
textBox1.Text = objReader.ReadToEnd();
```

Once opened, a StreamReader should be closed. This is quite simple:

```
objReader.Close();
```

After adding the **close** line, your code should look like this (again, this is the shorter XP version):

```
private void btnOpen_Click(object sender, EventArgs e)
{
    string file_name = "C:\\\\test1.txt";

    System.IO.StreamReader objReader;
    objReader = new System.IO.StreamReader(file_name);

    textBox1.Text = objReader.ReadToEnd();

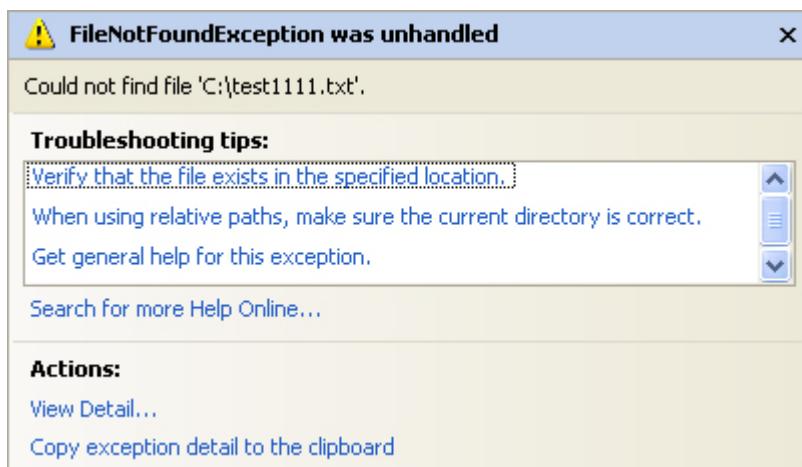
    objReader.Close();
}
```

Run your programme and test it out. You should find that the contents of your text file appear in your text box when your button is clicked.

Stop your programme and return to your code. Change your file_name line to this:

```
string file_name = "C:\\\\test1111.txt";
```

In other words, just add three more 1's to the file name. Run your programme and click the button again. Unless you have a file called **test1111.txt** on your C drive, you should see this error message appear (Visual Studio 2012 users will see a plainer version of the error below):



You can test to see if a file exists. Again, you use the System.IO namespace. But this time you use the File class. Here's the code:

```

if( System.IO.File.Exists( file_name ) == true )
{
    //OPEN FILE HERE
}
else
{
    //ERROR MESSAGE HERE
}

```

So the line that checks to see if the file exists is this:

System.IO.File.Exists(file_name) == true

Exists is a method available to **File**. Between the round brackets of **Exists**, you type the name of the file you want to check. After a pair of double equals signs, we're checking for a value of **true**.

Place the code that opens the file between the curly brackets of the IF statement. Add an error message for the **else** part, and we have this:

```

private void btnOpen_Click(object sender, EventArgs e)
{
    string file_name = "C:\\\\test1111.txt";

    if (System.IO.File.Exists(file_name) == true)
    {
        System.IO.StreamReader objReader;
        objReader = new System.IO.StreamReader(file_name);

        textBox1.Text = objReader.ReadToEnd();

        objReader.Close();
    }
    else
    {
        MessageBox.Show("No such file " + file_name);
    }
}

```

So the file gets checked to see if it exists. If it doesn't, we display an error message. If it does, the file is opened.

You can also check to see if a folder (Directory) exists. Just use **Directory** instead of **File**. Like this:

```

if( System.IO.Directory.Exists( folder_location ) )
{
}

```

Read a file line by line

Quite often, you'll want to read the contents of a text file line by line, as part of some processing operation. You can then move on to the next line and process that before moving on.

To do that, you need the **ReadLine** method of the **StreamReader** object. But it's not quite as straightforward as the **ReadToEnd** method.

Along with the **ReadLine** method, you use the **Peek** method in a **Do** loop. It sounds complicated, but examine this code:

```
do
{
    textLine = objReader.ReadLine() + "\r\n";
}

} while (objReader.Peek() != -1);
```

Inside of the **Do** loop, we have this:

```
textLine = objReader.ReadLine() + "\r\n";
```

The **ReadLine** method grabs one line from your text file. It's in a loop so that we can grab all the lines, one line at a time. We're just handing the line to a string variable called **textLine**. (The "**\r\n**" at the end adds a carriage return and a new line.)

The **while** part of the loop is this:

```
while (objReader.Peek() != -1);
```

The **Peek** method checks one character at a time. If it doesn't find a character it will return a value of minus 1. In which case, we can end the loop. As long as **Peek** returns a value other than minus one, we keep looping.

Try it out for yourself. Add another button to your form. Double click it to get at the coding window. Add the following code to your new button. (We've missed the error checking off to simplify things. And the **file_name** part is from the shorter XP version):

```
private void btnOpen_Click(object sender, EventArgs e)
{
    string file_name = "C:\\test1.txt";
    string textLine = "";

    System.IO.StreamReader objReader;
    objReader = new System.IO.StreamReader(file_name);

    do
    {
        textLine = textLine + objReader.ReadLine() + "\\r\\n";

    } while (objReader.Peek() != -1);

    textBox1.Text = textLine;

    objReader.Close();
}
```

Inside the Do loop, you can write whatever code you need to process the line of text. But all we're doing is building up the **txtLine** variable. After the loop ends, we're placing it all in the text box.

Before running your programme, though, add another few lines of text to your text file.

Write to a Text File

To write to a text file, you use the StreamWriter instead of the StreamReader. It's used in the same way, though:

```
System.IO.StreamWriter objWriter;
objWriter = new System.IO.StreamWriter(file_name);
```

Here, we're setting up a StreamWriter object and calling it **objWriter**. When you create a new StreamWriter object, you hand it the name of a file between the round brackets. Note that if the file is not found, no error will be raised. This is because a file is created if one doesn't exist.

Add another button to your form. Double click and type the following code for it (If you have a Windows version later than XP you can add the longer **file_name** lines, the ones with **Environment.GetFolderPath**, etc.):

```

private void btnWrite_Click(object sender, EventArgs e)
{
    string file_name = "C:\\\\test1.txt";

    System.IO.StreamWriter objWriter;
    objWriter = new System.IO.StreamWriter(file_name);

    objWriter.WriteLine(textBox1.Text);
    objWriter.Close();

    MessageBox.Show("Wrote File");
}

```

Notice the line the does the writing:

```
objWriter.WriteLine(textBox1.Text);
```

After the dot, use the **Write** method. In between the round brackets of **Write**, add what it is you want to write to your file. For us, this was the Text in **textBox1**.

We also close the **StreamWriter** object, after we have finished with it.

If you want to write line by line, instead of all at once, use this:

```
objWriter.WriteLine();
```

In between the round brackets of **WriteLine**, you add the line that you want to write to your text file. This is normally done in a loop, and is useful if you have an array of text that you want to write to a file. Here's a coding example that does just that. See if you can work out what's happening. Try it yourself.

```

private void btnOpen_Click(object sender, EventArgs e)
{
    string file_name = "C:\\\\Users\\\\Owner\\\\Documents\\\\testData.txt";
    string tLine = "";

    System.IO.StreamWriter objWriter;
    objWriter = new System.IO.StreamWriter(file_name);

    string[] aryTest = new string[5];

    aryTest[0] = "Mary";
    aryTest[1] = "had";
    aryTest[2] = "a";
    aryTest[3] = "little";
    aryTest[4] = "one";

    for (int i = 0; i < 5; i++)
    {
        tLine = tLine + aryTest[i] + "\\r\\n";
        objWriter.WriteLine(aryTest[i]);
    }

    textBox1.Text = tLine;
    MessageBox.Show("Wrote File");
    objWriter.Close();
}

```

The **for** loop is where we write each line to the text file, adding a carriage return and new line character.

Appending text to a file

When you use either **Write** or **WriteLine**, it will overwrite the current file. So if you click your button twice, it won't add the new text to the end. It will just erase whatever is already there, and write it again.

If you want to add more text to what you already have, then you need to append. This is quite easy, and is done with the StreamWriter line:

```
objWriter = new System.IO.StreamWriter( file_name, true );
```

Between the round brackets of StreamWriter, and after the file you want to use, you type **true** to append data to your file. The default is **false**.

And that's it - no need for anything else!

Copy a File

You can copy a file quite easily. It's done with the File class of System.IO. Let's see how. First create a new folder on your C drive. Give it the name **copiedFiles**. Add a new button to your form. Double click to get at the code, and set up the following files and directory paths:

```
string fileToCopy = "C:\\test1.txt";
string newLocation = "C:\\copiedFiles\\test1.txt";
string folderLocation = "C:\\copiedFiles";
```

In versions later than XP, you may have to change the file paths to your **Documents** folder:

```
string fileToCopy = "C:\\Users\\Owner\\Documents\\test1.txt"
string newLocation = "C:\\Users\\Owner\\Documents\\copiedFiles\\test1.txt";
string folderLocation = "C:\\Users\\Owner\\Documents\\copiedFiles";
```

So the file we want to copy is **test1.txt** and it is in the root folder of the C drive. We want to copy it to a new location. The folder (Directory) we want to copy it to is called **copiedFiles**. We added a Folder Location, because we want to check if this Directory exists.

To copy a file, you use the Copy method of the File class:

```
System.IO.File.Copy( fileToCopy, newLocation, true );
```

In between the round brackets of Copy, you need a file to copy and the new location of the file you're trying to copy. You can also set an overwrite option to

true or false. If the file being copied is already in the folder, setting an override value to true will replace it. The default is false, and you'll get an error if the file being copied already exists.

When the error checking is done as well, our code looks like this:

```
private void btnCopy_Click(object sender, EventArgs e)
{
    string fileToCopy = "C:\\\\test1.txt";
    string newLocation = "C:\\\\copiedFiles\\\\test1.txt";
    string folderLocation = "C:\\\\copiedFiles";

    if (System.IO.Directory.Exists(folderLocation))
    {
        if (System.IO.File.Exists(fileToCopy))
        {
            System.IO.File.Copy(fileToCopy, newLocation, true);
            MessageBox.Show("File Copied");
        }
        else
        {
            MessageBox.Show("No such File");
        }
    }
    else
    {
        MessageBox.Show("No such Directory");
    }
}
```

The code checks to see if the Directory exists. If it does, then we check to see if the file exists. If all is OK, then we go ahead and copy the file, setting overwrite to **true**.

Add the code to your own and try it out. When you click your button, you should see the “File Copied” message box appear. If you look in the **copiedFiles** folder that you created, you should see the **test1.txt** file there.

Move a File

To move a file to a new location, you would use the Move method of the File class:

```
System.IO.File.Move( fileToMove, fileLocation );
```

Everything else is the same: type a file to move between the round brackets of the **Move** method. Then, after a comma, add the new location.

Don't forget to add your error checking as well!

Delete a File

To delete a file from your computer, you can use the Delete method of the File class:

```
System.IO.File.Delete( file_path );
```

In between the round brackets of **Delete**, you need the name and path of the file you are trying to get rid of. Be very careful when trying this one out because it really does delete it. You won't find the file in the Recycle bin.

Conclusion

File manipulation is a very useful skill for you to master as a programmer. There's an awful lot more to it than in this book, obviously. But we've given you more than enough power to be getting on with!

In the next section, we'll take a look at databases and C#.

SQL Server Express and Visual C# .NET

In this section, you'll learn how to create a database with SQL Server Express. You can do all this within the Visual C# .NET software. Once you have a database, you'll learn how to pull records from it, and display them on a Windows Form. You will also learn how to navigate through the records in your database, and how to add new records.

What is SQL Server Express?

SQL Server Express is database system from Microsoft. It's the stripped down version of SQL Server, which is used by a lot of big businesses around the world. Fortunately, Microsoft have made the Express Edition a free download.

Hopefully, you downloaded SQL Server Express with the Visual C# .NET Express Edition. If you did, you should see this somewhere on your start menu:

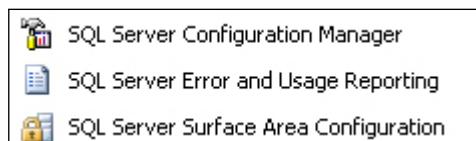


If you can't see the SQL Server Express entry, then you may need to download it. At the time of writing, the download location is here on the Microsoft site:

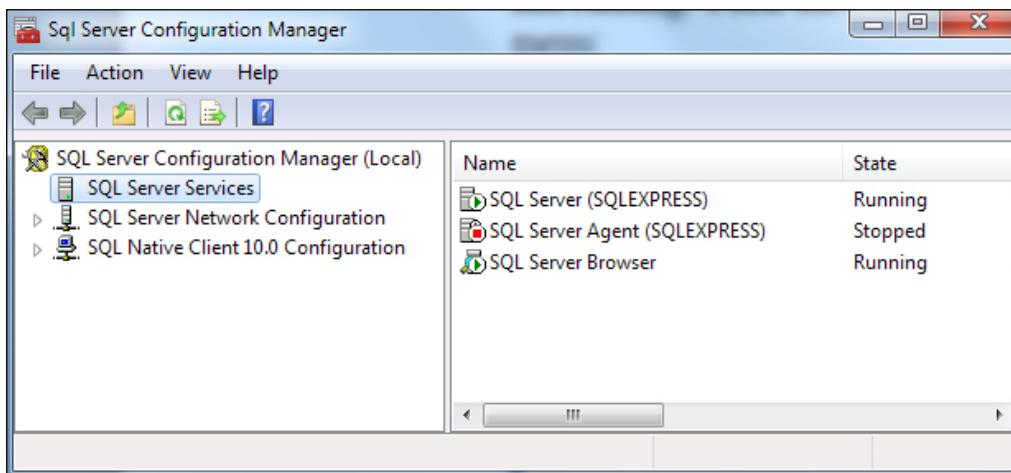
<http://www.microsoft.com/sqlserver/en/us/editions/2012-editions/express.aspx>

The SQL Server Management Studio Express you can see in the image above is a separate download, and is piece of software that lets you create and manage your databases without starting up your Visual C# .NET software. You don't need to download it for this book. But if you want to grab it, go to the Microsoft site and type "SQL Server Management Studio Express" into the search box.

Once you have downloaded and installed SQL Server Express, make sure it's up and running. Click the Configuration Tools menu above to see the following menu:



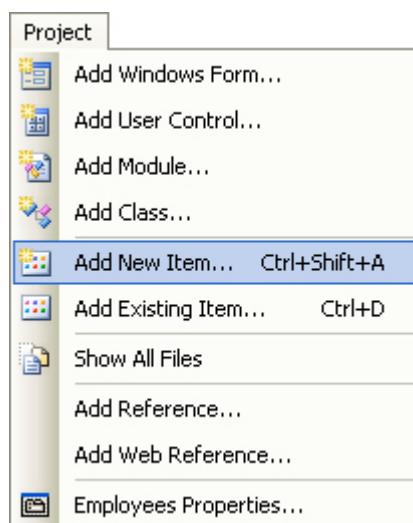
Select SQL Server Configuration Manager and you should see a window appear:



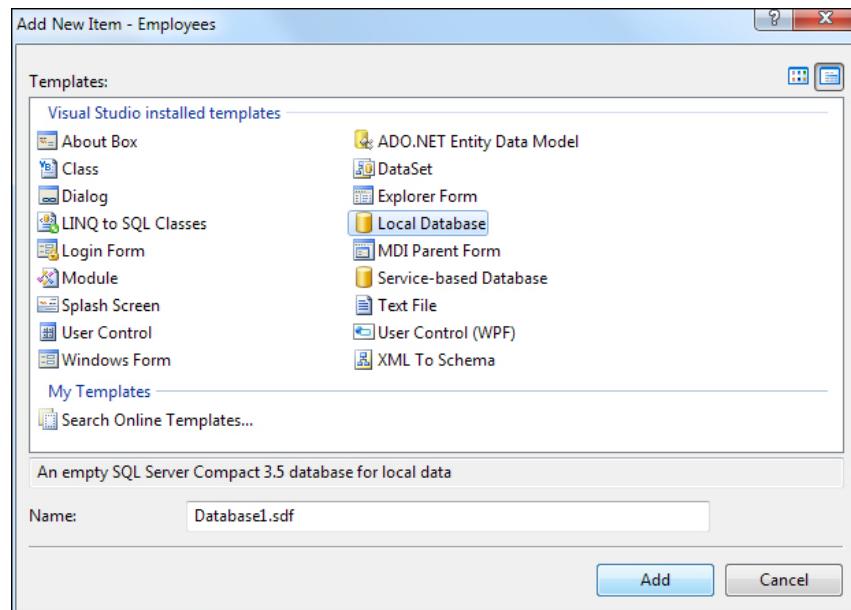
In the right pane, make sure the **State** for both says **Running**. If not, right click and select Start from the menu. You can then close the Configuration Manager.

How to Create a Compact SQL Server Database

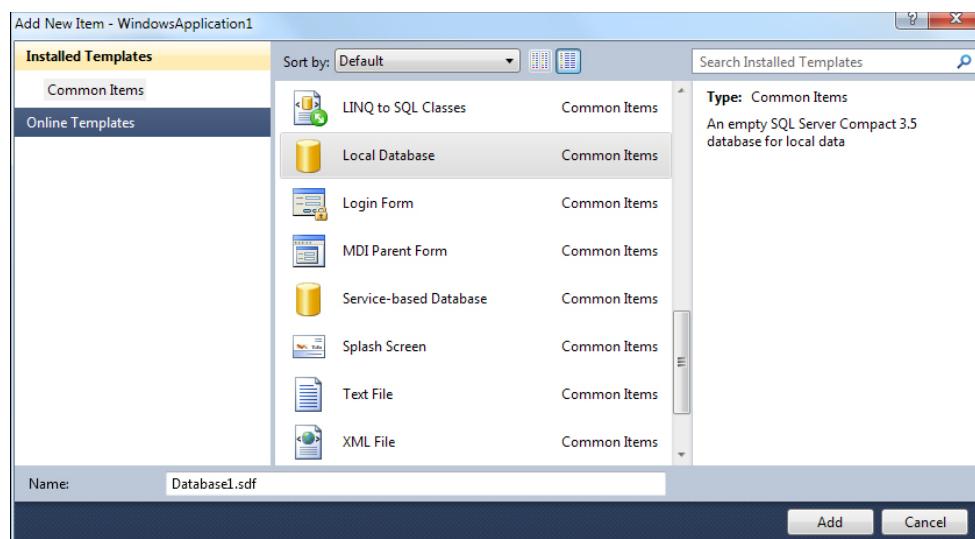
Start a new project, and call it anything you like as we're only using this project to create a database. From the menu at the top, click on **Project > Add New Item**.



From the **Add New Item** dialogue box in version 2008, look for the **Local Database** item, as in the image below:



In C# 2010, you'll see this dialogue box (the 2012 version is similar only less colourful):



Again, select **Local Database**.

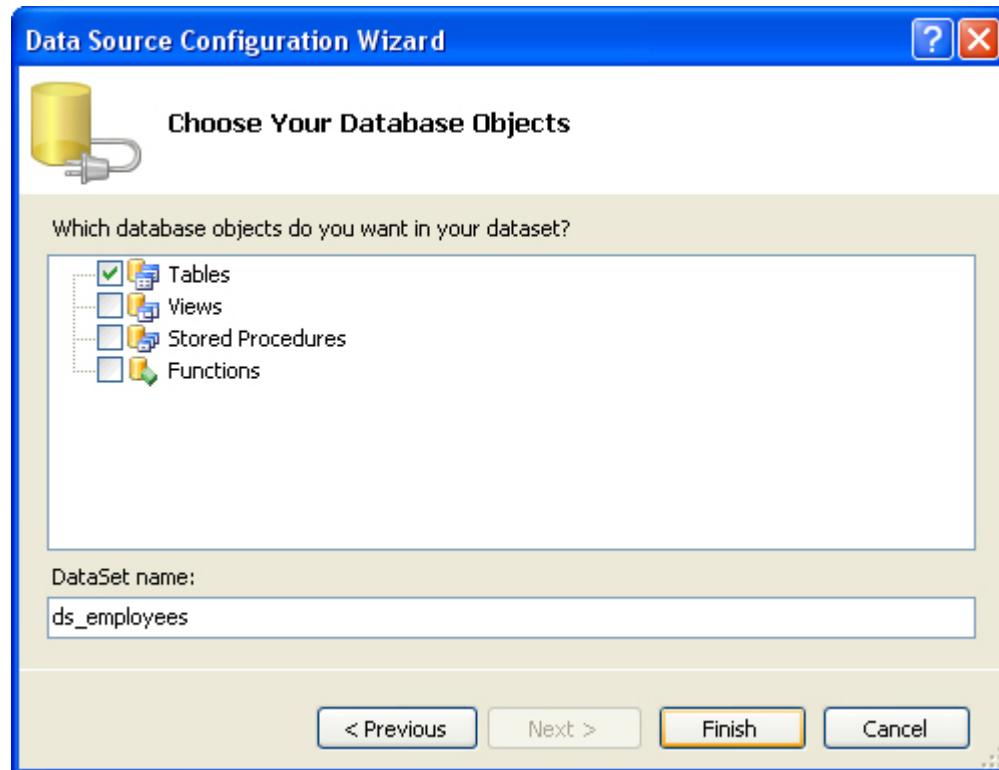
If it's not there, then you may not have SQL Server Express installed. In which case, you need to download it. At the time of writing, the location is:

<http://www.microsoft.com/sql/editions/express/default.mspx>

If you see the Local Database item, click on it to select it. Note the file ending for a SQL Server Compact database, though – it's an SDF file. Change the name to this:

Employees.sdf

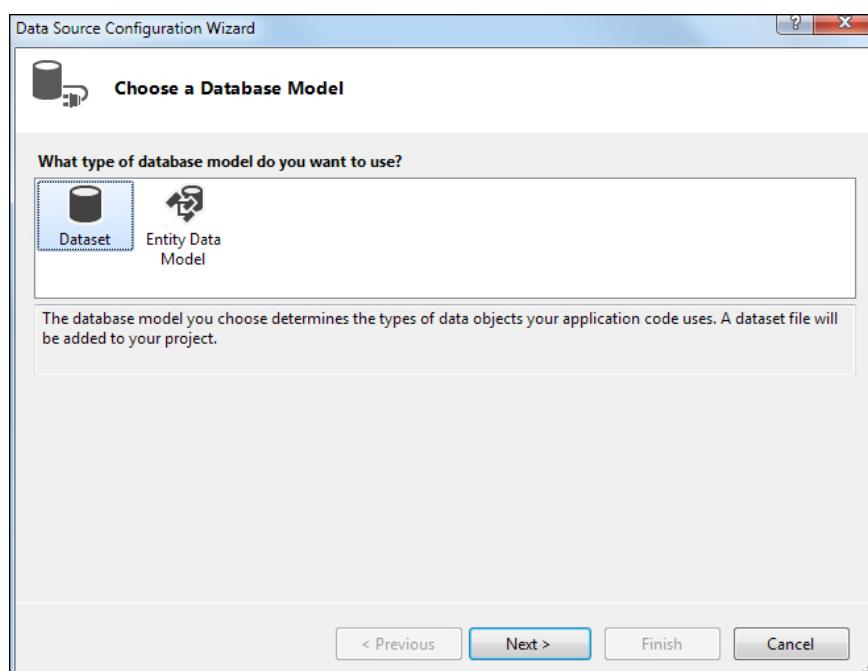
Now click the **Add** button. You should, eventually, see the Data Source Configuration Wizard appear. This one in version 2008:



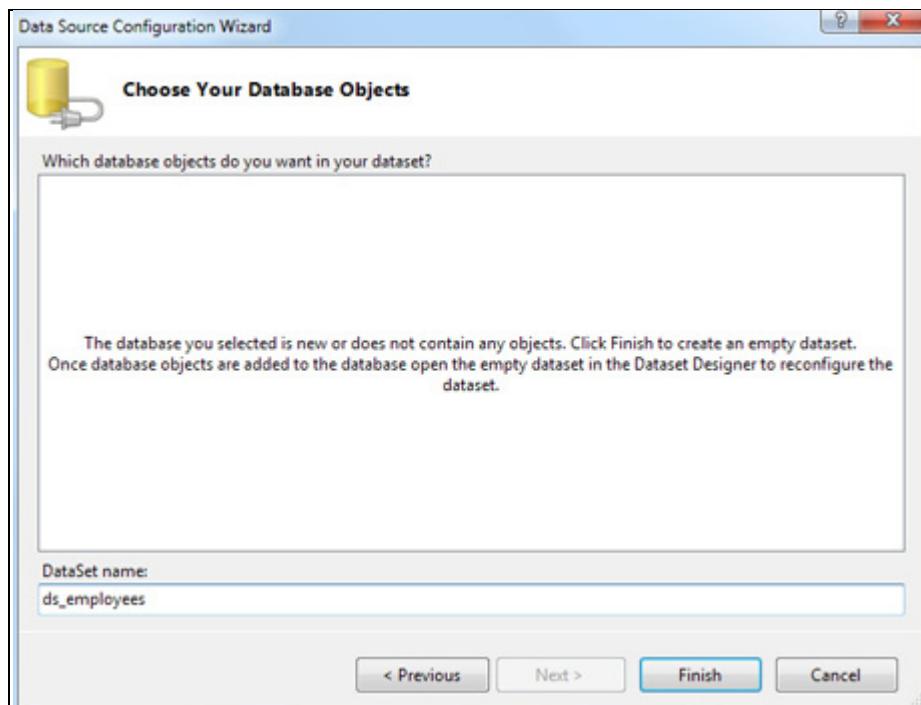
Put a tick in the box next to Tables, and then change the name of the dataset to this:

ds_employees

In version 2010 and 2012 of the software, you'll see this instead:



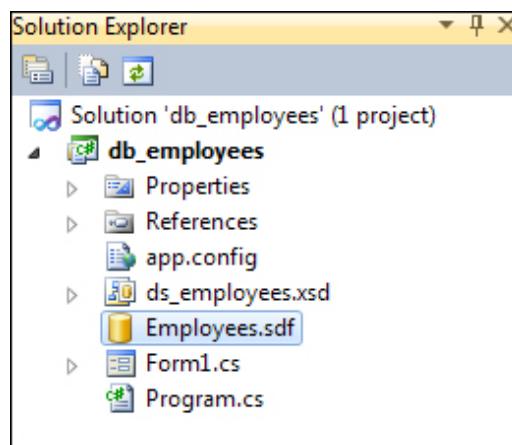
Select **Dataset**, then click **Next**. You'll then see this:



Again, change the name of the Data Set to **ds_employees**.

Click the Finish button.

It might look as though nothing has happened, after you click the Finish button. But have a look at the Solution Explorer at the top right of your screen. You should see your database there:

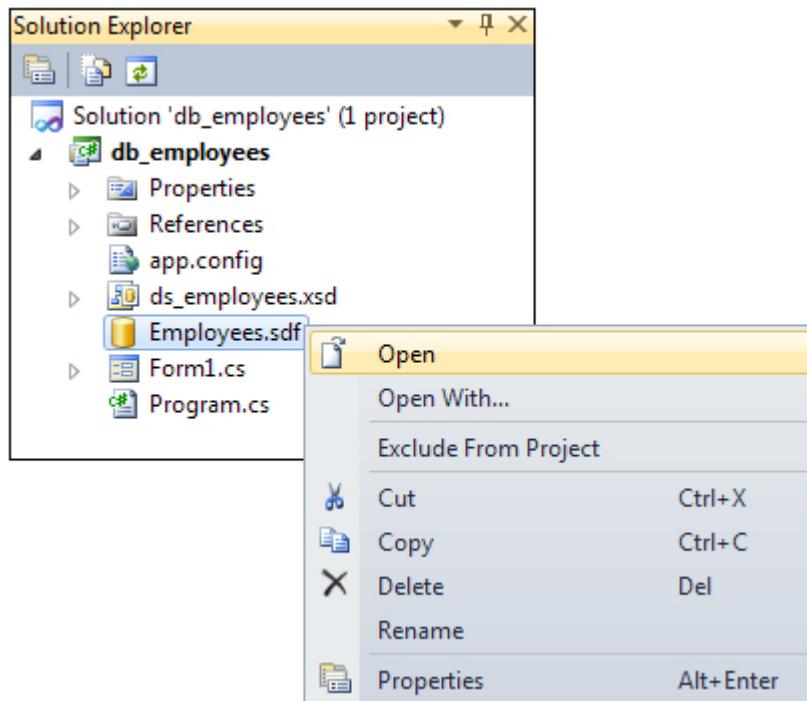


When you save your project, the database will get saved along with all the solution files.

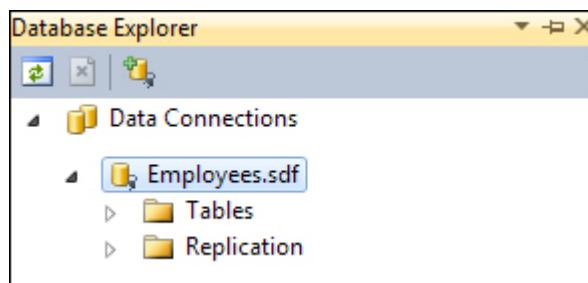
How to Create Tables in your SQL Server Database

Now that you have created the database itself, you need to create at least one table to go in it.

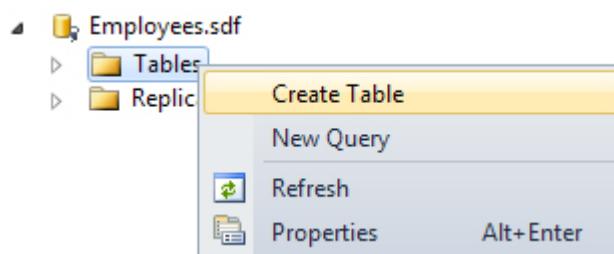
To create a table, right click on the database in the Solution Explorer. From the menu that appears, select **Open**:



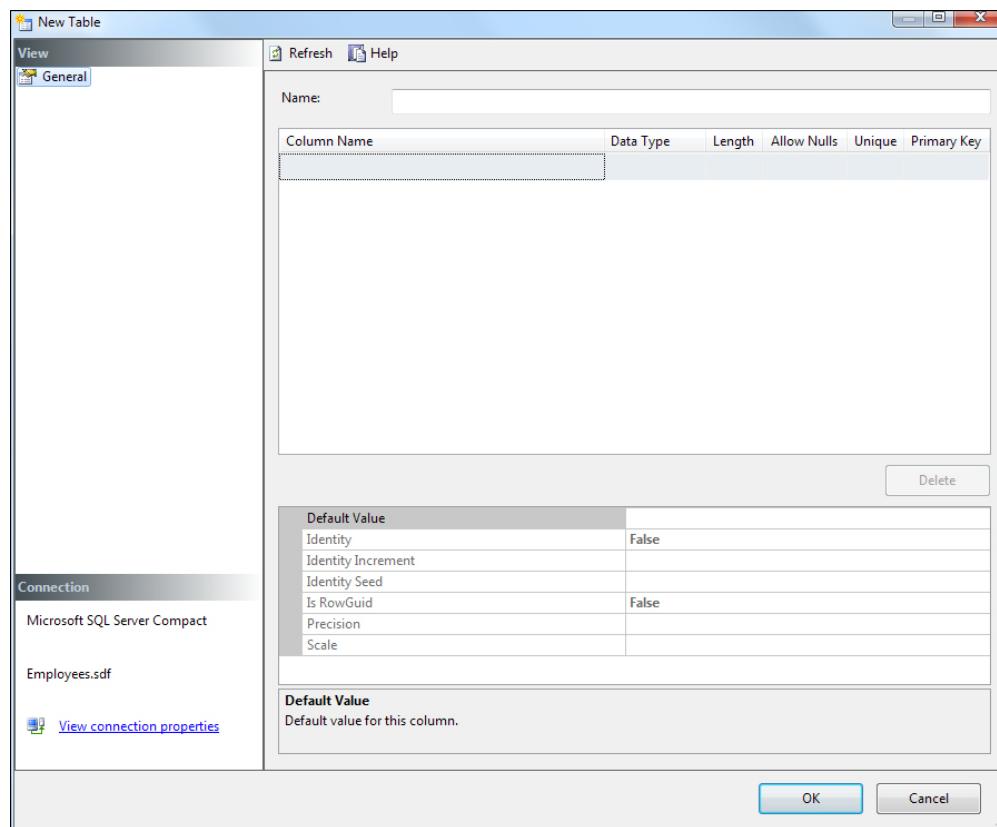
When you click **Open**, you'll see the Database Explorer appear on the left hand side of your screen:



To create a new table, right click on **Tables**. From the menu that appears, select **Create Table (Add New Table in versions 2008)**:



When you click **Create Table**, you'll see a dialogue box appear. This one:



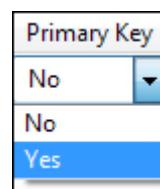
This screen is where you set up the Columns that go in the table. You'll get to enter the actual data later. But you need to tell SQL Server what kind of data (**Data Type**) is going into each column.

First, though, type a name for your table in the **Name** box at the top. Call the table **tbl_employees**.

For the first column in the table, type **ID** in the **Column Name** box. You'll then see some more information appear:

| Name: tbl_employees | | | | | | |
|----------------------------|-----------|--------|-------------|--------|-------------|--|
| Column Name | Data Type | Length | Allow Nulls | Unique | Primary Key | |
| ID | nvarchar | 100 | Yes | No | No | |

We'll set the ID column as a Primary Key. Click into the **Primary Key** box and select **Yes** from the dropdown list:



For the other values, set the **Data Type** to **int**, the **Length** to 4, **Allow Nulls** to **No**, and **Unique** to **Yes**. Your first column should then look like this:

| Column Name | Data Type | Length | Allow Nulls | Unique | Primary Key |
|-------------|-----------|--------|-------------|--------|-------------|
| ID | int | 4 | No | Yes | Yes |
| | | | | | |

The Primary Key field, the ID column, will need to have a new integer assigned every time a new entry is added to the table. In other databases, this is known as an auto increment number. So that this happens automatically without us having to worry about it, have a look at the bottom of your Table screen. You should see a Properties area. Click on **Identity** and set it to **True**:

| | |
|--------------------|-------|
| Default Value | |
| Identity | True |
| Identity Increment | 1 |
| Identity Seed | 1 |
| Is RowGuid | False |
| Precision | |
| Scale | |
| | |

The Increment value is set to 1, meaning the database itself will automatically add 1 when a new record is added.

Using the same technique as above, create the following Column Names, Data Types, Length Allow Nulls, Unique and Primary Key values:

| Column Name | Data Type | Length | Allow Nulls | Unique | Primary Key |
|-------------|-----------|--------|-------------|--------|-------------|
| ID | int | 4 | No | Yes | Yes |
| first_name | nvarchar | 50 | Yes | No | No |
| last_name | nvarchar | 50 | Yes | No | No |
| job_title | nvarchar | 50 | Yes | No | No |
| department | nvarchar | 50 | Yes | No | No |

If you're not too sure about database terminology by the way, here are what all those headings and values mean:

Data Type – This is the kind of data going into a table column. The ID column can only have integers, while all the others can only have text (nvarchar).

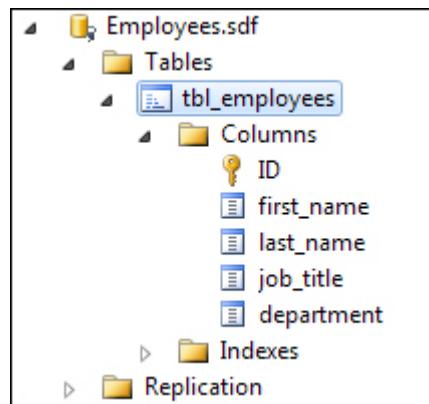
Length – How many characters or digits the field can hold.

Allow Nulls – A Null value is one where nothing is set for that field, not even a zero or a blank string.

Unique – This means whether or not you can have duplicate values in this field

Primary Key - This is used for things like linking tables together, indexing a long table, and for searching.

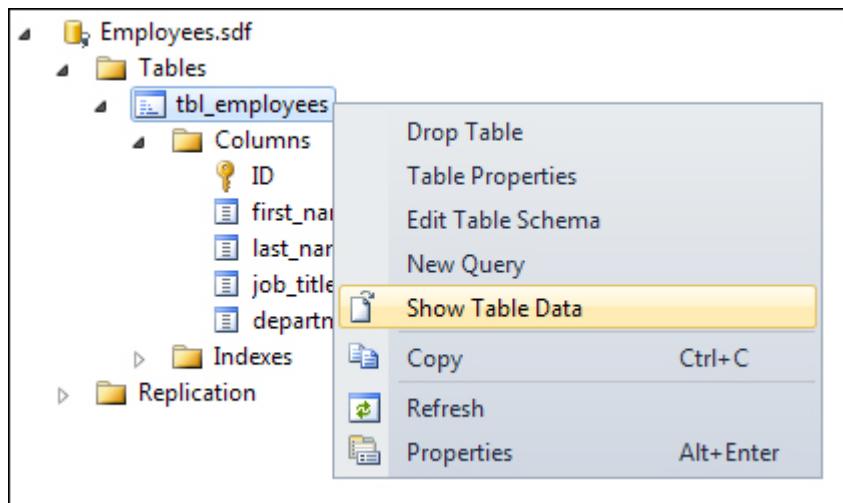
Click OK when you've finished adding all the information to the New Table dialogue box. Your Database Explorer should show you the new table you've just added:



Now that you've set up the table, you can start adding data to it.

Adding Data to a SQL Server Database Table

To add entries to your SQL Server database, right click on your table name in the Database explorer. From the menu that appears, select **Show Table Data**:



When you select **Show Table Data**, you'll see a new tab appear:

| | ID | first_name | last_name | job_title | department |
|---|------|------------|-----------|-----------|------------|
| * | NULL | NULL | NULL | NULL | NULL |

All our Column names are there, waiting to be filled in. To enter data, simply click inside a cell and start typing.

Click inside the **first_name** column. (The ID Column will take care of itself.) Type a first name. Click inside of **last_name** and type a last name. Click inside of **job_title** and enter a job title. Then enter the department. You can enter the same details as ours, if you prefer (all made up):

| | ID | first_name | last_name | job_title | department |
|---|------|------------|-----------|-----------------|------------|
| * | NULL | Adara | Hussein | Lead Program... | IT |

Notice the warning symbols in the cells. These appear when the cell data has changed. The ID is still NULL in the image above. When we click in the next row, however, notice that a number will appear in the first ID cell:

| | ID | first_name | last_name | job_title | department |
|---|------|------------|-----------|-----------------|------------|
| * | 1 | Adara | Hussein | Lead Program... | IT |
| * | NULL | NULL | NULL | NULL | NULL |

The reason it does this is because we set **Is Identity** to True, and the **Identity Increment** to 1 – it's an Auto Increment field, in other words.

But we have now created one row in our database table. Fill out a few more rows. You can use the same details as ours, in the image below:

| | ID | first_name | last_name | job_title | department |
|---|------|------------|-----------|-------------------|------------|
| | 1 | Adara | Hussein | Lead Programmer | IT |
| | 2 | Bibi | Saleem | Head Writer | Creative |
| | 3 | Hamal | Ata | Network Engineer | IT |
| | 4 | Haris | Hameed | Systems Analyst | IT |
| | 5 | Tansy | Lakshman | Writing Assistant | Creative |
| | 6 | Orenda | Khan | Head of Tuition | Teaching |
| | 7 | Tadi | Patel | Network Engineer | IT |
| | 8 | Zoe | Walker | Head of Design | Creative |
| | 9 | Alice | Thyne | Tutor | Teaching |
| ▶ | 10 | Jake | Jaloore | Graphic Artist | Creative |
| * | NULL | NULL | NULL | NULL | NULL |

Don't worry if your numbers for the ID column are not sequential (are not 1 to 10). If you make a mistake and delete a Row, you are given the next number after the deleted Row, and not the next number in your sequence.

Save your work, and you will have created your very first Compact SQL Server Express database! But it's a huge subject, and whole books have been written about SQL Server. We can only touch on the very basics here. What we do have, though, is a database we can open with C# .NET programming code. We'll do that next.

Create a Database Project

Close the entire Project down (**File > Close Project or Close Solution**), because we're going to be copying the database we just created. We're doing this so that we don't run into any problems with this error:

“An attempt to attach an auto-named database ...”

To find your database, have a look in your Visual Studio Projects folder, usually located in the **Documents** folder. Find the folder with the name of the Project you just closed down. You should see the **Employees.sdf** file. If you can't find it that way, do a search through the Windows Start menu.

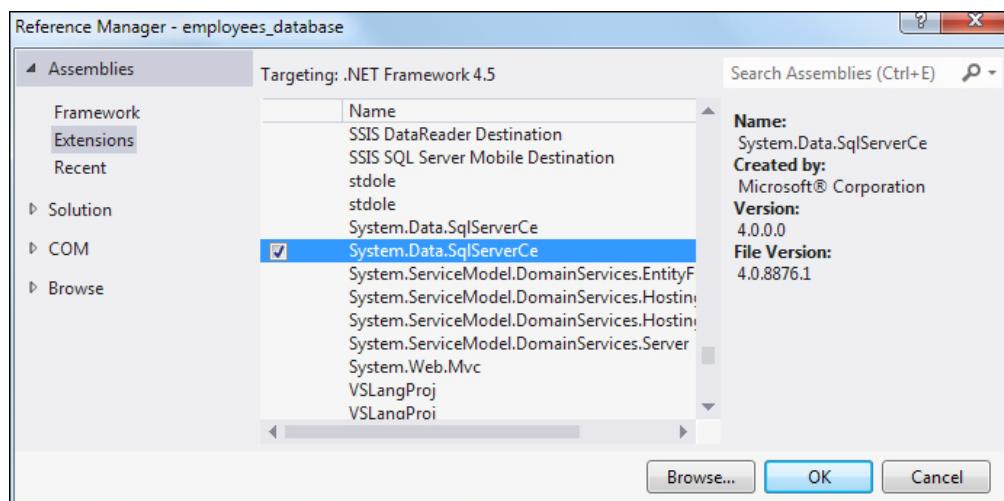
Once you've found the database, copy the SDF file to the Documents folder (My Documents in Windows XP). You can create a new folder for this. Call it databases. Copy your database to this new folder.

Now that you have moved the database, create a New project in C# (**File > New Project**). Give it the name **employees_database**.

Connecting to a SQL Server Express Database

To connect to a database using SQL Server Express, you first need to set up a SQL Connection object. You then need something called a connection string to tell C# where the database is.

If you're using a Compact SQL Server database (Local Dataase) then the easiest connection object to use is the **SqlCeConnection** object. If you have Visual Studio Express 2012, however, you need to add a reference in order to get the various **SqlCeConnection** objects. From the menu bar at the top of Visual Studio Express 2012 click **Project**. From the Project menu, select **Add Reference**. You'll then see the following dialogue box appear:



On the left hand side, click on the **Extensions** item, you'll then see a list of objects you can add to your project. Scroll down to near the bottom and you'll see an item for **System.Data.SqlServerCe**. You may see two of these, a later version and an earlier version. Hold your mouse over the latest version and you'll see a checkbox appear. Put a check in the checkbox and then click OK. You will now be able to use the **SqlCeConnection** objects.

To set up a **SqlCeConnection** connection variable, double click the blank form. Just outside of the Form Load event add the following:

```
System.Data.SqlServerCe.SqlCeConnection con;
```

When you type a dot after SqlServerCe you'll see lots of SqlCe objects appear. We're using the SqlCeConnection object and giving it the name con.

Inside of the Form Load event, we can create a new con object. Add the following line:

```
con = new System.Data.SqlServerCe.SqlCeConnection();
```

A connection object needs a connection string. This points the connection object to where your database is. Add the following line to your code:

```
con.ConnectionString = "Data Source=C:\\\\Users\\\\Owner\\\\Documents\\\\Employees.sdf";
```

ConnectionString is a Property of connection objects. After an equal sign, you type the path to your database file. The path needs to go after the text "Data Source=". Notice where all the double quotes are. Notice, too, that we've used two backslash characters. The backslash character is considered a special character in C# programming, so it needs to be escaped. To escape a backslash character (or any other character) just type another backslash before it.

To try to open a connection, you need to issue the **Open** command:

```
con.Open( );
```

Next, add message box to see if everything is working OK:

```
MessageBox.Show("Connection Open");
```

If you have opened a connection, then you need to close it:

```
con.Close( );
```

Once you've added the lines above, your code should look like this:

```
System.Data.SqlServerCe.SqlCeConnection con;

private void Form1_Load(object sender, EventArgs e)
{
    con = new System.Data.SqlServerCe.SqlCeConnection();
    con.ConnectionString = "Data Source=C:\\\\Users\\\\Owner\\\\Documents\\\\databases\\\\Employees.sdf";
    con.Open();

    MessageBox.Show("open");

    con.Close();
}
```

Run your programme and you should see the message box appear when you click your button.

Stop your programme and return to your code. You can delete or comment out the message box.

Datasets and Data Adapters

The connection to the database has been made. The next step is to pull the records from our Employees table. To do that, a **Dataset** and a **DataAdapter** are needed.

A **Dataset** is where all your data is held when it is pulled from the database table. Think of it like a grid that you see on a spreadsheet. The Columns in the grid are the Columns from your database table. The Rows represent a single entry in the table.

The Dataset needs to be filled with data. However, because the Dataset and Connection object can't see each other, they need someone in the middle to help them out – the **DataAdapter**. The DataAdapter will fill the Dataset with records from the database.

So we need to set up two more objects, a Dataset and a DataAdapter. To create a Dataset object, add the following just above the form load event:

```
DataSet ds1;
```

Inside of the form load event, create a new object from the Dataset type we've called **ds1**:

```
ds1 = new DataSet();
```

Add the code just after your **con.Open** line.

For the DataAdapter, add the following outside of the form load event:

```
System.Data.SqlServerCe.SqlCeDataAdapter da;
```

We're setting up a DataAdapter variable, here, and calling it **da**.

Inside of the form load event, we can create a new object from our **da** variable. Add these two lines just after your **ds1** line:

```
string sql = "SELECT * From tbl_employees";
da = new System.Data.SqlClient.SqlDataAdapter( sql, con );
```

The first line sets up a string variable called **sql**. SQL stands for Structured Query Language. It's a language used to pull records from a database, and variants of it are used for all database systems. You use the Structured Query Language on the database itself. (SQL Server's variant is called T-SQL. The T stands for Transact.)

Keywords in SQL are SELECT, UPDATE, WHERE, and a whole lot more besides. The * symbol means “all the records”. So we're saying, “Select all the records from the table called **tbl_employees**”.

The DataAdapter object will use your SQL commands to pull the records from the database. But you need to tell it which connection object to use. That's why, in between the round brackets, we have this:

```
( sql, con );
```

Our new DataAdapter object will then know what records to pull (sql), and where to pull them from (con).

But here's what your coding windows should look like:

```
System.Data.SqlServerCe.SqlCeConnection con;
System.Data.SqlServerCe.SqlCeDataAdapter da;
DataSet ds1;

private void Form1_Load(object sender, EventArgs e)
{
    con = new System.Data.SqlServerCe.SqlCeConnection();
    con.ConnectionString = "Data Source=C:\\\\Users\\\\Owner\\\\Documents\\\\databases\\\\Employees.sdf";

    con.Open();

    ds1 = new DataSet();
    string sql = "SELECT * FROM tbl_employees";
    da = new System.Data.SqlServerCe.SqlCeDataAdapter(sql, con);

    MessageBox.Show("open");

    con.Close();
}
```

To fill the dataset with records from the database, you use the DataAdapter and issue the **Fill** command:

```
da.Fill( ds1, "Workers" );
```

What this does is to **Fill a Dataset** called **ds1**. After the comma, you can type an identifying name for this particular Fill. We've called ours "Workers".

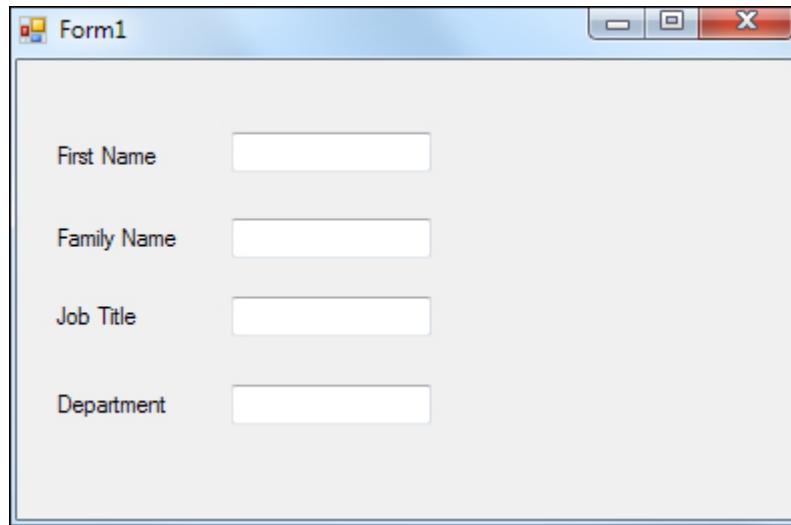
After the Fill command has been issued, the records from the SQL command are stored in the Dataset. This, remember, is just like a grid with Columns and Rows.

So add the line to your code. Put it just before the **con.Close()** line.

Accessing Data from the Dataset

At the moment, we have a Dataset filled with records from the database table. But we can't actually see anything. What we'd like to do is to display the records on a form. We'll put the data in textboxes.

Add four text boxes and four labels to your form, so that the design looks something like ours below:



When the form loads, we want the first record from the Dataset to appear in the text boxes.

We'll do all that from a method. So just after your form load code, add a new method called **NavigateRecords**. It's not going to return a value, so you can make it a void method.

Trying to get at the data from a Dataset can be a torturous business, because there are so many properties and methods to access. The easiest way is to first set up a new **DataRow** variable:

```
DataRow dRow;
```

This will refer to a row from the Dataset:

```
DataRow dRow = ds1.Tables["Workers"].Rows[0];
```

So after the equals sign, we have this:

```
ds1.Tables["Workers"].Rows[0];
```

You first type the name of your Dataset, which is **ds1** for us. After a dot, select **Tables** from the IntelliSense list. **Tables** is a collection, and stores a list of all the available Tables (a Table is just that grid that we mentioned). To tell C# which Table you want, type its name between square brackets and a pair of double quotes. After another dot, select Rows from the IntelliSense list. In between square brackets, you specify which Row from the Dataset you want. Row zero [0] is the first Row in the Table.

So add that line to your **NavigateRecords** method.

But that's not the end of it! We've only pointed to a Row in the Dataset. We also need to specify a column.

To get at a Column in the Row, the code is this:

dRow.ItemArray.GetValue(1).ToString()

We've started with our Row object, which we've called **dRow**. After a dot, select **ItemArray** from the IntelliSense list. This is an Array of all the Items (Columns) in your Row. We had five columns in our database table: ID, first_name, family_name, job_title, and department. ItemArray starts at zero, so ID will be Item 0, first_name will be Item 1, last_name will be Item 2, job_title will be Item 3, and department will be Item 4.

After another dot, then, select **GetValue** from the list. As its name suggests, this will Get the Values from your Columns. In between round brackets, you need the Item number from the array. **GetValue(1)** will refer to the first_name column in our Dataset. Finally, you need to convert it to a string with **ToString()**. Once converted to a string, you can put it straight into a text box.

Putting all that together, add the following code to your **NavigateRecords** method:

```
private void NavigateRecords()
{
    DataRow dRow = ds1.Tables["Workers"].Rows[0];
    textBox1.Text = dRow.ItemArray.GetValue(1).ToString();
    textBox2.Text = dRow.ItemArray.GetValue(2).ToString();
    textBox3.Text = dRow.ItemArray.GetValue(3).ToString();
    textBox4.Text = dRow.ItemArray.GetValue(4).ToString();
}
```

If you prefer, you can put everything on one line. But it will be a very long line. Here it is:

```
textBox1.Text = ds1.Tables["Workers"].Rows[0].ItemArray.GetValue(1).ToString();
```

But you are almost ready to test it all out. The final thing to do is to add a call to your **NavigateRecords** method. Put the call just before the **con.Close** line, as in the image below:

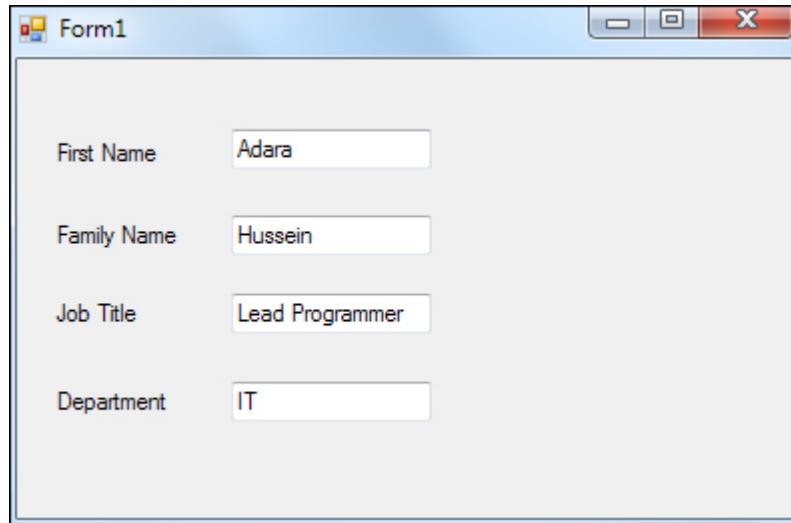
```
da.Fill(ds1, "Workers");
NavigateRecords();

con.Close();

}

private void NavigateRecords()
{
    DataRow dRow = ds1.Tables["Workers"].Rows[0];
    textBox1.Text = dRow.ItemArray.GetValue(1).ToString();
    textBox2.Text = dRow.ItemArray.GetValue(2).ToString();
    textBox3.Text = dRow.ItemArray.GetValue(3).ToString();
    textBox4.Text = dRow.ItemArray.GetValue(4).ToString();
}
```

Now you can test it out. Run your programme and the form should display the first record from your database. It should look like ours:



Now that we have one record displayed, we can add buttons to navigate backwards and forward through all the records in our database.

Database Navigation Buttons

The first thing we'll do is to allow users to move forward through each record in our database. This is done with just a bit of programming logic, and manipulating the Row value in the Dataset.

To make this work, we need to set up a few variables. So return to your coding window, and add the following two variables outside the form load event, just below the three you already have:

```
int MaxRows = 0;
int inc = 0;
```

Your coding window will then look like this:

```
System.Data.SqlServerCe.SqlCeConnection con;
System.Data.SqlServerCe.SqlCeDataAdapter da;
DataSet ds1;

int MaxRows = 0;
int inc = 0;

private void Form1_Load(object sender, EventArgs e)
{
```

The MaxRows variable will hold how many Rows there are in the Dataset. This is so that we don't go past the last record when the **Next Record** button is clicked. If we try to go past the last record, the programme will crash! (We'll add the button shortly.)

The **inc** variable will be used to change the current Row number.

To get at the number of Rows in the DataSet, you can use the **Count** property of Rows. Add this code to your form load event, just below your call to **NavigateRecords()**:

```
MaxRows = ds1.Tables["Workers"].Rows.Count;
```

Instead of specifying a particular Row in square brackets, this time we type a dot, and then select **Count** from the IntelliSense list. This will return how many rows there are in this particular Dataset.

When the form loads, then, MaxRows will contain a count of how many Rows are in the Dataset called **ds1**.

For the **NavigateRecords** method, we need to make one slight change. At the moment, we have this code:

```
DataRow dRow = ds1.Tables["Workers"].Rows[0];
```

But this will point to Row[0] all the time. We can use the **inc** variable here. What we'll do is to increment the value when the Next Record button is clicked, adding 1 to **inc** every time.

Change the line to this:

```
DataRow dRow = ds1.Tables["Workers"].Rows[inc];
```

The only change is in between the square brackets of Rows.

Run your programme to test if it works. You should still see the first record displayed in your text boxes.

Stop your programme and return to the design environment. Add a button to your form. Change the Text property to **Next Record**. Change the Name property to **btnNext**.

Double click your button to get at the coding window. For the code, we need to check what is inside of the MaxRows variable and make sure we don't go past it. We also need to increment the **inc** variable. It is this variable that will move us on to the next record.

Add the following if statement to your button:

```

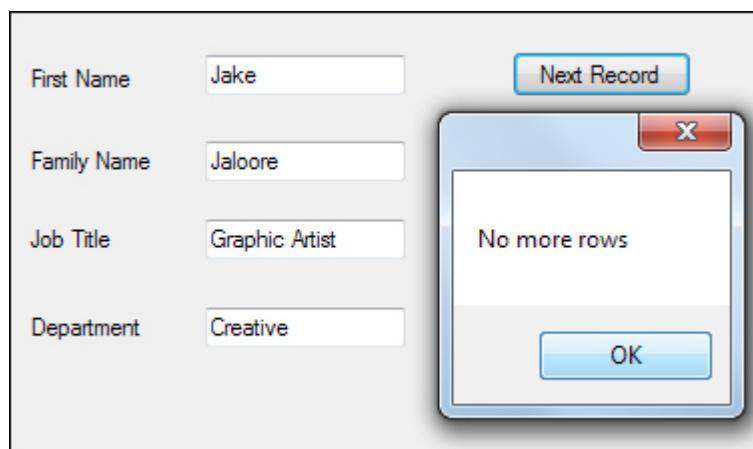
private void btnNext_Click(object sender, EventArgs e)
{
    if (inc != MaxRows - 1)
    {
        inc++;
        NavigateRecords();
    }
    else
    {
        MessageBox.Show("No more rows");
    }
}

```

The first line of the If Statement says “If **inc** does not equal **MaxRows** minus 1”. If it doesn’t then we increment the **inc** variable and call **NavigateRecords**. But can you see why we need to say **MaxRows - 1**? It’s because of the **Rows[inc]** line in our **NavigateRecords** method. The count for Rows starts at zero. So if we only have 4 records in the database, the count will be for 0 to 3. **MaxRows**, however, will be 4. If we don’t deduct 1, the programme will crash with an error: **IndexOutOfRangeException**.

If the **MaxRows** is reached, then we can display a message for the user.

Run your programme and test it out. You should be able to move forward through your database. Here’s what your form should look like when the last record is reached:



Move Backwards through the Database

We can use similar code to move backwards through the records in the database. Add another button to your form. Change the Text property to **Previous Record**. Change the Name property to **btnPrevious**.

Double click your new button to get at the coding window. Now add the following:

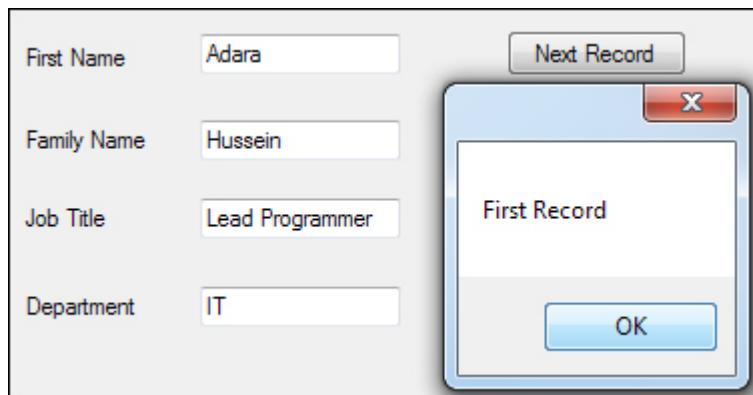
```

if (inc > 0)
{
    inc--;
    NavigateRecords();
}
else
{
    MessageBox.Show("First Record");
}

```

The if statement is now only checking the **inc** variable. We need to check if it's greater than zero. If it is, we can deduct 1 from inc, and then call our **NavigateRecords** methods. When the form loads, remember, inc will be 0. So if we tried to move back one record after the form first loads the programme would crash. It would crash because we'd be trying to access Rows[-1].

Run your programme and test it out. Click you Previous Record button and you should see this:



Click both of your buttons and make sure you can move back and forward through the records. You programme shouldn't crash!

Jump to the Last Record in your Database

To move to the last record of your database, you only need to make sure that the inc variable and MaxRows have the same value.

Add a new button to your form. Set the Text property as **Last Record**, and the Name property as **btnLast**. Double click, and add the following code:

```

if (inc != MaxRows - 1)
{
    inc = MaxRows - 1;
    NavigateRecords();
}

```

The If Statement again checks that inc is not equal to MaxRows minus 1. If it isn't, we have this:

```
inc = MaxRows - 1;
```

MaxRows minus 1 would equal 3 in our four record database. Because Rows[inc] goes from 0 to 3, this is enough to move to the last record after the call to NavigateRecords.

Jump to the First Record in your Database

To move to the first record in the database, we only need to set **inc** to zero.

Add another button to your form. Change the Text property to **First Record**. Change the Name property to **btnFirst**. Double click your new button and add the following code:

```
if (inc != 0)
{
    inc = 0;
    NavigateRecords();
}
```

This just checks to see if inc isn't already zero. If it isn't, we set the inc variable to 0. Then we call the NavigateRecords method.

Run your programme and test it out. You should now be able to move through the records in your database without the programme crashing. What we'll do now is to allow the user to add a new record to the database. This is more complex than the navigation, so you may need to pay close attention!

Add a New Record

When you add a new record, you'll want to add it to the Dataset and the underlying database. Let's see how.

Add two new buttons to the form. Set the following properties for your buttons:

| | |
|--------------|-----------|
| Name: | btnAddNew |
| Text: | Add New |
| | |
| Name: | btnSave |
| Text: | Save |

The Add New button won't actually add a new record. The only thing it will do is to clear the text boxes, ready for a new record to be added. The Save button is where we'll add the record to the Dataset and to the Database.

Double click your Add New button, and add code to clear the text boxes:

```
textBox1.Clear();
textBox2.Clear();
textBox3.Clear();
textBox4.Clear();
```

That's all we need to do here. You can test it out, if you want. But all the code does is to clear the four text boxes of text. The user can then enter a new record.

After a new record has been entered into the text boxes, we can Save it. So double click your Save button to get at the code.

To save a record, you need to do two things: save it to the Dataset, and save it to the underlying database. You need to do it this way because the Dataset with its copy of the records is disconnected from the database. Saving to the Dataset is NOT the same as saving to the database.

To add a record to the Dataset, you need to create a new Row:

```
DataRow dRow = ds1.Tables["Workers"].NewRow();
```

This creates a New DataRow called dRow. But the Row will not have any data. To add data to the row, the format is this:

```
dRow[1] = textBox1.Text;
```

So after your DataRow variable (**dRow**) you need a pair of square brackets. In between the square brackets type its position in the Row. This is the Column number. **dRow[1]** refers to the **first_name** column, for us. After an equals sign, you type whatever it is you want to add to that Column – the text from **textBox1**, in our case.

Finally, you issue the Add command:

```
ds1.Tables["Workers"].Rows.Add( dRow );
```

After Add, and in between a pair of round brackets, you type the name of the Row you want to add, which was **dRow** in our example. The new Row will then get added to the end of the Dataset.

So add this code to your Save button:

```

private void btnSave_Click(object sender, EventArgs e)
{
    DataRow dRow = ds1.Tables["Workers"].NewRow();

    dRow[1] = textBox1.Text;
    dRow[2] = textBox2.Text;
    dRow[3] = textBox3.Text;
    dRow[4] = textBox4.Text;

    ds1.Tables["Workers"].Rows.Add(dRow);

    MaxRows = MaxRows + 1;
    inc = MaxRows - 1;
}

```

Notice the last two lines:

```

MaxRows = MaxRows + 1;
inc = MaxRows - 1;

```

Because we have added a new Row to the Dataset, we also need to add 1 to the MaxRows variable. The inc variable can be set to the last record in the Dataset.

Try it out. When you start your programme, click the Add New button to clear the text boxes. Enter a new record in the blank text boxes and then click your Save button. Click your Previous and Next buttons. You'll see that the new record appears.

(Obviously, you'll want to add error checking code to check that the Save button is not clicked before the Add button. Or simply set the Enabled property to false for the Save button when the form loads. You can then set Enabled to true in your Add button.)

If you close the programme down, and start it back up again you'll find that the new record has disappeared! It's disappeared because we haven't yet added it to the underlying database. We've only added it to the Dataset.

To add a new record to the Database, you need to use something called a Command Builder. You use the Command Builder in conjunction with a DataAdapter. This has an Update method that will do the job for you. The only thing you need to do is tell it which Dataset and table holds all the records. Let's see how.

Add the following line to your Save button: (Add it after ds1.Tables but before the MaxRows line)

```
System.Data.SqlServerCe.SqlCeCommandBuilder cb;
```

This sets up a SqlCeCommandBuilder variable. We've called it cb. Next, you can create an object from your variable:

```
cb = new System.Data.SqlServerCe.SqlCeCommandBuilder( da );
```

Notice that the SqlCeCommandBuilder now has a pair of round brackets. In between the round brackets you type the name of a DataAdapter. We set one up at the top of the code called **da**, so we can use this.

The Command Builder has its own DataAdapter, which in turn has an Update method. The next line to add, therefore, is this:

```
cbDataAdapter.Update( ds1.Tables["Workers"] );
```

In between the round brackets of update, you need a dataset. You then refer to the table associated with the dataset. For us this was **ds1.Tables["Workers"]**.

Your code for the Save button, then, should look like this:

```
private void btnSave_Click(object sender, EventArgs e)
{
    DataRow dRow = ds1.Tables["Workers"].NewRow();

    dRow[1] = textBox1.Text;
    dRow[2] = textBox2.Text;
    dRow[3] = textBox3.Text;
    dRow[4] = textBox4.Text;

    ds1.Tables["Workers"].Rows.Add(dRow);

    System.Data.SqlServerCe.SqlCeCommandBuilder cb;
    cb = new System.Data.SqlServerCe.SqlCeCommandBuilder(da);
    cbDataAdapter.Update(ds1.Tables["Workers"]);

    MaxRows = MaxRows + 1;
    inc = MaxRows - 1;

    btnAddNew.Enabled = true;
    btnSave.Enabled = false;
}
```

Notice that we've disabled the Save button but enabled the AddNew button. We did the reverse with the AddNew button:

```
textBox1.Clear();
textBox2.Clear();
textBox3.Clear();
textBox4.Clear();

btnAddNew.Enabled = false;
btnSave.Enabled = true;
```

You can try your programme out, now. Click your **Add New** button and add a new record. Click the Save button. When you close the programme down and reopen it, the new record should be there.

Update a Record

Sometimes, all you want to do is to update a record in the database. This is very similar to Adding a new record. Examine the following code:

```
private void UpdateDB()
{
    System.Data.SqlClient.SqlCommandBuilder cb;
    cb = new System.Data.SqlClient.SqlCommandBuilder(da);
    cbDataAdapter.Update(ds1.Tables["Workers"]);
}

private void btnUpdate_Click(object sender, EventArgs e)
{
    DataRow dRow2 = ds1.Tables["Workers"].Rows[inc];

    dRow2[1] = textBox1.Text;
    dRow2[2] = textBox2.Text;
    dRow2[3] = textBox3.Text;
    dRow2[4] = textBox4.Text;

    UpdateDB();

    MessageBox.Show("Data Updated");
}
```

The first thing to notice is that we've set up a method called **UpdateDB**. This has all the code for the Command Builder and is doing the actual updating. We can then remove this same code from the **Save** button:

```
private void btnSave_Click(object sender, EventArgs e)
{
    DataRow dRow = ds1.Tables["Workers"].NewRow();

    dRow[1] = textBox1.Text;
    dRow[2] = textBox2.Text;
    dRow[3] = textBox3.Text;
    dRow[4] = textBox4.Text;

    ds1.Tables["Workers"].Rows.Add(dRow);

    UpdateDB();

    MaxRows = MaxRows + 1;
    inc = MaxRows - 1;

    btnAddNew.Enabled = true;
    btnSave.Enabled = false;
}
```

Notice the call to our new method:

```
UpdateDB( );
```

For the Update button, the first line of the code is this:

```
= ds1.Tables["Workers"].Rows[inc];
```

The only thing you're not doing is adding a new Row. After creating a new Row called dRow2, we set it to the current Row, using our **inc** variable. Whatever is in the text boxes then gets transferred to dRow2[1], dRow2[2], dRow2[3] and dRow2[4]. These are the Columns in the Row.

When you run your form again, amend one of your records. Close down the form and open it back up again. You should find that your amendments are still there.

Delete a Record

To delete a record from the Dataset, you use the **Delete** method:

```
ds1.Tables["Workers"].Rows[inc].Delete( );
```

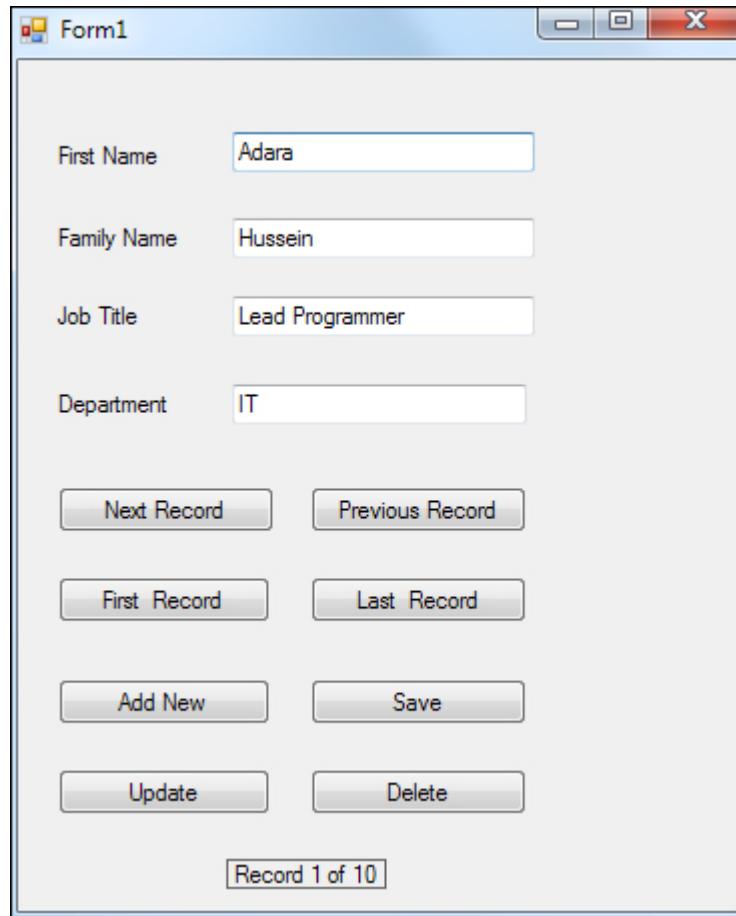
This is enough to Delete the entire Row (**Rows[inc]**). But it is only deleted from the Dataset. Here's the code to delete the record from the database, as well:

```
private void btnDelete_Click(object sender, EventArgs e)
{
    ds1.Tables["Workers"].Rows[inc].Delete();
    UpdateDB();
    MaxRows = ds1.Tables["Workers"].Rows.Count;
    inc--;
    NavigateRecords();
    MessageBox.Show("Record Deleted");
}
```

After the **Delete()** line, we make a call to our **UpdateDB** method again. This will commit the changes to the database. Because we've deleted a record, we need to reset the **MaxRows** count, because it will be one less. We also need to deduct one from the **inc** variable. After this, we can make a call to **NavigateRecords**. The final line just displays a message that the record was deleted.

Exercise P

Examine this version of our form on the next page:



If you look at the bottom, you'll see a label that says "Record 1 of 10". Implement this in your own programme. If you set up a method, you can just call it when the form loads, and again from NavigateRecords.

[Answer to Exercise P](#)

Finding Records

A useful feature to add is a Find button. When a find button is clicked, you then display the record that the user was searching for. Or display a “Not found” message if there were no matching records.

So add a new button to your form. Set the Text property to **Find**, and the Name property to **btnFind**. Double click your button to get at the coding window.

There are quite a few different ways you can implement a search. The method we'll use is to Select a row from the dataset. We'll allow a user to search using a last name.

Add the following three lines to your **btnFind** code:

```
string searchFor = "Ata";
```

```
int results = 0;
DataRow[] returnedRows;
```

The first variable sets up a string called **searchFor**. This is obviously the record we want to find. We've hard-coded the value, here, and just entered a last name from our database table. But you'd want this value to come from a text box on your form.

The second variable, results, will be used to tell us whether or not any results were found.

The third line is a DataRow array, which we've called returnedRows. We're using an array because more than one record might be found. Each record will then be stored in a position in the array.

To get at a particular Row in your Dataset, you can use the Select method. Here's the code. It's a bit long, so we've had to spread it over two lines. It should be one line in your code:

```
returnedRows = ds1.Tables["Workers"].Select("last_name=' " + searchFor + "'');
```

So you start with your Dataset, which was **ds1** for us. Then you need the name of a Table in your Dataset. We want to search the "Workers" table. After a dot, we have the **Select** method:

```
Select("last_Name=' " + searchFor + "'');
```

It looks a bit messy with all those quote marks. But first we have an outer pair:

```
Select(" ");
```

Inside of these two double quotes, we have this:

```
last_name=
```

You need to type the name of a Column from your Dataset, here. We're using the **last_name** Column. But we could have used the **first_name** Column instead:

```
first_name=
```

The Column names are the same ones we used in our database table. But notice the equals sign. After that, you need a value to search for. This needs to go in single quotes. So if you wanted to hard-code a **last_name** value it would be this:

```
Select("last_Name='Ata'");
```

We're using a variable name (**searchFor**), instead.

The Select method allows you to use other SQL keywords. If you don't want an exact search, for example, you can use Like instead of =.

```
Select("last_name Like 'Ata'")
```

Note where the single quotes are – surrounding the text you want to search for. Because our search used a variable, we're using plus symbols to concatenate. Which is why it's so messy!

If a row is found, it will then be stored in the **returnedRows** array. To get a count of how many rows were found, we can use this code:

```
results = returnedRows.Length;
```

This just uses the Length property of the **returnedRows** array. The length is how many items are in the array. If it's greater than zero, it means we've found a match. We can use an if statement to check:

```
if (results > 0)
{
    //RECORD FOUND
}
else
{
    MessageBox.Show("No such Record");
}
```

If a record is found, we need to get at the values in the Columns. We can create a new Row for this:

```
if (results > 0)
{
    DataRow dr1;

    dr1 = returnedRows[0];
}
```

We now set up a DataRow called dr1. We want the first returned Row to be stored here. The first Row is returnedRows[0];

Putting it all together, here's the full code for the search:

```

private void btnFind_Click(object sender, EventArgs e)
{
    string searchFor = "Ata";

    int results = 0;
    DataRow[] returnedRows;

    returnedRows = ds1.Tables["Workers"].Select("last_name='" + searchFor + "'");

    results = returnedRows.Length;

    if (results > 0)
    {
        DataRow dr1;

        dr1 = returnedRows[0];

        MessageBox.Show(dr1[1].ToString() + " " + dr1[2].ToString());
    }
    else
    {
        MessageBox.Show("No such Record");
    }
}

```

Notice the line that displays a message:

MessageBox.Show(dr1[1].ToString() + " " + dr1[2].ToString());

Because **dr1** is now a **DataRow**, you can access its data by either using the Column name, or the index number. So these lines return the same values:

**dr1["job_title"]
dr1["first_name"]
dr1["last_name"]**

**dr1[3]
dr1[1]
dr1[2]**

It's up to you which ones you want to use.

But try your programme out. Click your Find button and a search result should display.

Close your programme down. Change the name of the person being searched for and try again.

Exercise Q

Add a text box to your form. Get the name of the person from this text box, rather than using the hard coded value that you have at the moment.

[Answer to Exercise Q](#)

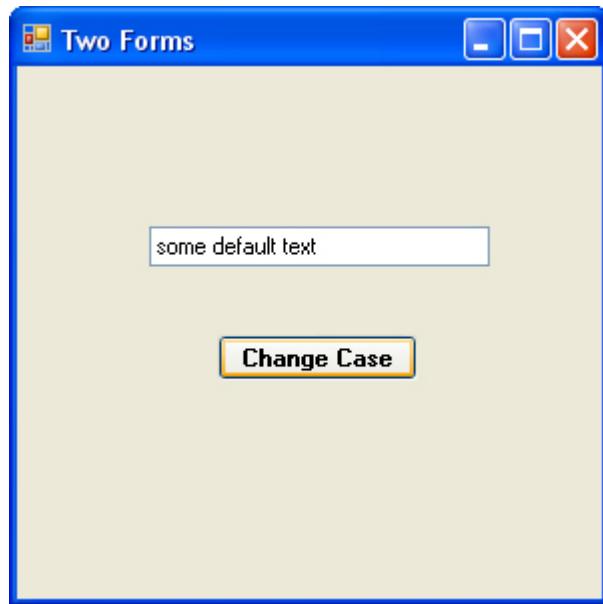
OK, that's enough of databases! It's a huge subject, obviously, and many books have been written on the subject. We've only touched the surface in these lessons, and encourage you to delve deeper. Especially if you want a job as a programmer! In the next and final section, we'll take a look at some other things you can do with C# .NET. Up first are multiple forms.

Creating Multiple Forms

There aren't many programmes that have only one form. Most programmes have other forms that are accessible from the main one that loads at start up. In this section, you'll learn how to create programmes with more than form.

The programme we'll create is very simple one. It will have a main form with a text box and a button. When the button is clicked, it will launch a second form. On the second form, we'll allow a user to change the case of the text in the text box on form one.

Here's what form one looks like:

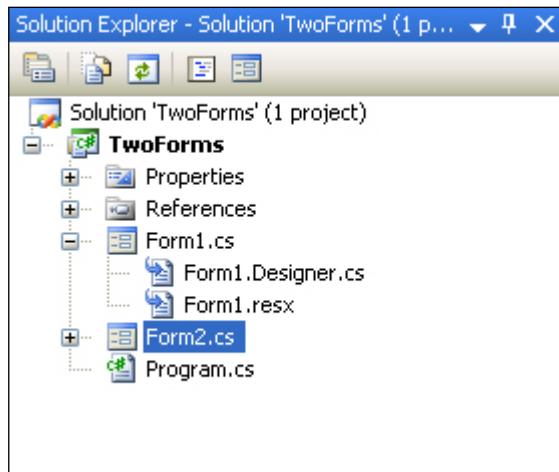


Design the above form. Set the Name property of the text box to **txtChangeCase**. For the Text property, add some default text, but all in lowercase letters. Set the Name property of the button to **btnFormTwo**.

Adding a new form to the project is easy. Click Project from the menu bar at the top of the Visual C# software. From the Project menu, select **Add Windows Form**. You'll see the Add New Item dialogue box appear. Make sure Windows Form is selected. For the Name, leave it on the default of **Form2.cs**. When you click OK, you should see a new blank form appear:



It will also be in the Solution Explorer on the right:



Adding the form to the project is the easy part – getting it to display is an entirely different matter!

To display the second form, you have to bear in mind that forms are classes. When the programme first runs, C# will create an object from your Form1 class. But it won't do anything with your Form2 class. You have to create the object yourself.

So double click the button on your Form1 to get at the coding window.

To create a Form2 object, declare a variable of Type Form2:

```
Form2 secondForm;
```

Now create a new object:

```
secondForm = new Form2();
```

Or if you prefer, put it all on one line:

```
Form2 secondForm = new Form2();
```

What we've done here is to create a new object from the Class called Form2. The name of our variable is **secondForm**.

To get this new form to appear, you use the `Show()` method of the object:

```
secondForm.Show();
```

Your code should now look like this:

```
private void btnFormTwo_Click(object sender, EventArgs e)
{
    Form2 secondForm = new Form2();
    secondForm.Show();
}
```

Run your programme and test it out. Click your button and a new form should appear – the blank second form.

However, there's a slight problem. Click the button again and a new form will appear. Keep clicking the button and your screen will be filled with blank forms!

To stop this from happening, move the code that creates the form outside of the button. Like this:

```
Form2 secondForm = new Form2();

private void btnFormTwo_Click(object sender, EventArgs e)
{
    secondForm.Show();
}
```

Try your programme again. Click the button and you won't get lots of forms filling the screen.

Modal Forms

Return to the code for your button. Instead of using the Show method, change it to this:

```
secondForm.ShowDialog();
```

The method we're now using is **ShowDialog**. This creates what's known as a Modal form. A Modal form is one where you have to deal with it before you can continue. Run your programme to test it out. Click the button and a new form appears. Move it out of the way and try to click the button again. You won't be able to.

Modal forms have a neat trick up their sleeves. Add two buttons to your blank second form. Set the following properties for them:

Name: btnOK
Text: OK

Name: btnCancel
Text: Cancel

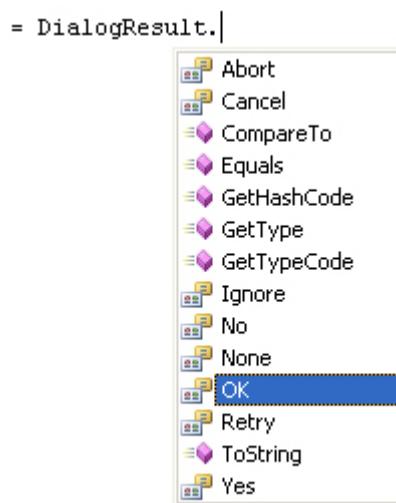
Your second form will then look like this:



Double click the OK button and add the following:

```
this.DialogResult = DialogResult.OK;
```

After you type the equals sign, the IntelliSense list will appear. Select DialogResult again, then a dot. The IntelliSense list will then show you this:



Select OK. What this does is to record the result of the button click, and set it to OK.

Double click your Cancel button and add the following code:

```
this.DialogResult = DialogResult.Cancel;
```

It's the same code, except we've chosen **Cancel** as the Result. Your coding window for form 2 should look like this:

```
public Form2()...  
  
private void btnOK_Click(object sender, EventArgs e)  
{  
    this.DialogResult = DialogResult.OK;  
}  
  
private void btnCancel_Click(object sender, EventArgs e)  
{  
    this.DialogResult = DialogResult.Cancel;  
}
```

You can use Form1 to get which of the buttons was clicked on Form2. Was it OK or was it Cancel?

Change the button code on Form1 to this:

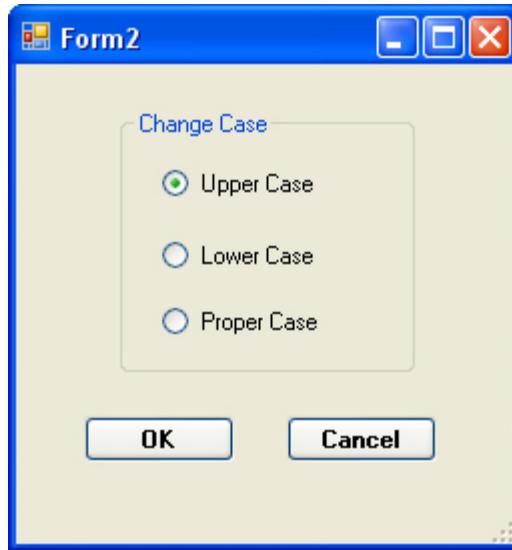
```
private void btnFormTwo_Click(object sender, EventArgs e)  
{  
  
    if (secondForm.ShowDialog() == DialogResult.OK)  
    {  
        MessageBox.Show("OK button clicked");  
    }  
}
```

The code checks to see if the OK button was clicked. If so, it displays a message. We'll get it to do something else in a moment. But you don't have to do anything with the Cancel button: C# will just unload the form for you.

Try it out. Click your Change Case button on Form1. When your new form appears, click the OK button. You should see the message. Try it again, and click the Cancel button. The form just unloads.

Getting at the values on other Forms

Turn your Form2 into a Change Case dialogue box, just like ours below:



When the OK button is clicked, we want the text in the text box on Form1 to change case, depending on which of the three options was chosen.

The problem we face is that the text box is private to Form1, and can't be seen from outside it. If you tried to refer to the text box from Form2, you'd just get errors.

One solution is to set up a public static variable, of type TextBox. You then assign textBox1 to this new variable.

So add the following to Form1:

```
public static TextBox tb = new TextBox();
```

This creates a new TextBox object called **tb**. Add the line just under your Form variable, and your coding window will look like this:

```
public Form1()...
```



```
Form2 secondForm = new Form2();
public static TextBox tb = new TextBox();

private void btnFormTwo_Click(object sender, EventArgs e)
{
    secondForm.ShowDialog();
}
```

Notice that we've deleted the message box code, and went back to the original. That's because we don't need the message box anymore. Delete yours as well.

Now that we have a TextBox object, we can assign our text box on form one to it. In the Form Load event of Form1, add the following line:

```
tb = txtChangeCase;
```

(The easiest way to bring up the code stub for the Form Load event is to double click a blank area of the form in design view.)

Here's what all the Form1 code looks like now:

```
public partial class Form1 : Form
{
    public Form1()
    {
        Form2 secondForm = new Form2();
        public static TextBox tb = new TextBox();

        private void btnFormTwo_Click(object sender, EventArgs e)
        {
            secondForm.ShowDialog();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            tb = txtChangeCase;
        }
    }
}
```

When the main form (Form1) loads, the text box will now be available to Form2.

So double click your OK button on Form2 to bring up its code stub. Enter the following:

```
string changeCase = Form1.tb.Text;
```

We're setting up a string variable called **changeCase**. The contents of this new string variable will be the **Text** from the text box called **tb** on **Form1**.

To change the case of the text, we can use string methods for two of them: Uppercase and Lowercase. Like this:

```
changeCase = changeCase.ToUpper();
```

```
changeCase = changeCase.ToLower();
```

Unfortunately, C# .NET does not have a direct string method to change text to Proper Case (or Title case as it's also known). Proper Case is capitalising the first letter of each word. For example, "This Is Proper Case".

In order to get Proper Case, you have to reference two System namespaces. One called Globalization and one called Threading. Add the following to the very top of the coding window:

```
using System.Globalization;
using System.Threading;
```

The code window will then look something like this:

Now that we have these two references, the next thing to do is to set up something called a `CultureInfo` object:

```
CultureInfo properCase = Thread.CurrentThread.CurrentCulture;
```

The `CurrentCulture` tells you information about the various language options of your particular country. Our `CultureInfo` object is called **properCase**.

That's not the end of it, though! You also need a `TextInfo` object:

```
TextInfo textInfoObject = properCase.TextInfo;
```

It's this `TextInfo` object that has the methods we need. We're setting up a `TextInfo` object called **textInfoObject**. We're handing it the `TextInfo` property of our **properCase** `CultureInfo` object.

Finally, we can change the case:

```
changeCase = textInfoObject.ToTitleCase( changeCase );
```

The `TextInfo` object has a method called **ToTitleCase**. In between the round brackets of the method, you type what it is you want to convert.

Hopefully, in future versions of C#, they'll add an easier way to convert to Proper Case!

To get which of the options on our `Form2` was chosen, we can add a series of **if ... else** statements:

```

if (radioButton1.Checked == true)
{
    changeCase = changeCase.ToUpper();
}
else if (radioButton2.Checked == true)
{
    changeCase = changeCase.ToLower();
}
else if (radioButton3.Checked == true)
{
    CultureInfo properCase = Thread.CurrentThread.CurrentCulture;
    TextInfo textInfoObject = properCase.TextInfo;

    changeCase = textInfoObject.ToTitleCase(changeCase);
}

```

So we're just checking to see which radio button was selected. We're then doing the case conversion.

To put the changed text into the text box on Form1, add the following line:

Form1.tb.Text = changeCase;

Add the line just before your DialogResult line. The full code for the button should be this:

```

private void btnOK_Click(object sender, EventArgs e)
{
    string changeCase = Form1.tb.Text;

    if (radioButton1.Checked == true)
    {
        changeCase = changeCase.ToUpper();
    }
    else if (radioButton2.Checked == true)
    {
        changeCase = changeCase.ToLower();
    }
    else if (radioButton3.Checked == true)
    {
        CultureInfo properCase = Thread.CurrentThread.CurrentCulture;
        TextInfo textInfoObject = properCase.TextInfo;

        changeCase = textInfoObject.ToTitleCase(changeCase);
    }

    Form1.tb.Text = changeCase;

    this.DialogResult = DialogResult.OK;
}

```

Run your programme and test it out. Click your button to bring up Form2. Select the Upper Case option and then click your OK button. You should find that the text in **txtChangeCase** on your main form will now be in uppercase.

Working with Dates and Time in Visual C# .NET

At some stage of your programming career, you'll need the ability to manipulate dates and time. A typical example would be a database programme where you want to record when an entry was made, especially if it's an order for a product. Or if you want to, say, calculate how many days it's been since an order was placed.

Start a new project for this. Add a button to your new form, and double click it to get at the code.

An inbuilt structure you can use to manipulate dates is called **DateTime**. Add this to your button code:

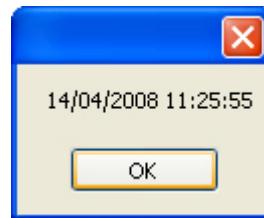
```
DateTime theDate;  
  
theDate = DateTime.Now;  
  
MessageBox.Show( theDate.ToString() );
```

After setting up a DateTime variable called **theDate**, we have this line:

```
theDate = DateTime.Now;
```

The Now property returns the current system date and time of your computer, wherever you are in the world.

The third line in our code converts the DateTime into a string and then displays it in a message box. When the code is run, and the button clicked, the message box will be something like this:



This is the date in UK format (day first, then month), followed by the current time (hours, minutes and seconds).

You can also have this, instead of Now:

```
theDate = DateTime.Today;
```

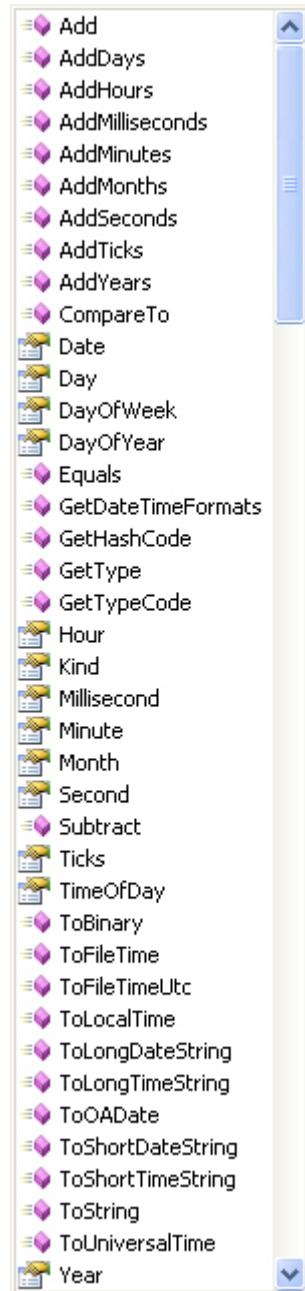
And even this

```
theDate = DateTime.UtcNow;
```

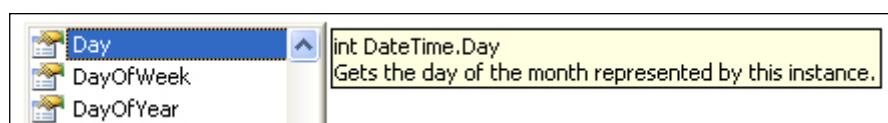
Utc is something called Coordinated Universal Time, or International Atomic Time.

Try all three in your code and see how they differ.

To get at just the year, or the month, or the day, take a look at the IntelliSense list when you type the dot after DateTime:



If you click on **Day** to highlight it, you'll see that it is an Integer:



To use this, then, you can set up a new integer variable and hand it the day:

```
theDate = DateTime.UtcNow;
```

```
int theDay = theDate.Day;
```

The Month and Year are also Integers, so the code is similar:

```
int theMonth = theDate.Month;
```

Or

```
int theYear = theDate.Year;
```

You can also convert your date to a string, and use something called a Format Provider. Try this code:

```
DateTime theDate = DateTime.UtcNow;
```

```
string custom = theDate.ToString("d");
```

```
MessageBox.Show(custom);
```

When you run your programme and click the button, the date displays in this format:

14/04/08

Change the “d” in the code above to “D” (capital D instead of lowercase). When you run the code, the date is displayed like this:

14 April 2008

Here's a list of letters you can use, and what they will display. Try a few and see for yourself:

| Format | Displays |
|----------|-------------------------------|
| d | 14/04/2007 |
| D | 14-Apr-07 |
| f | 14/04/2007 14:53 |
| F | 14/04/2007 14:53 |
| g | 14/04/2007 14:53 |
| G | 14/04/2007 14:53 |
| m | 14-Apr |
| r | Wed, 14 Jan 2007 17:53:30 GMT |
| s | 2007-04-14T17:53:30 |
| t | 14:53 |
| T | 14:53:30 |
| u | 2007-04-14 17:53:30Z |
| U | 14/04/2007 12:35 |
| y | Apr-14 |

Another thing you can do with DateTime is to specify a format. Type the following:

```
DateTime firstDate = new DateTime
```

After the final “e”, type a round bracket. You should see this:

```
DateTime firstDate = new DateTime()
▲ 1 of 12 ▾ DateTime.DateTime()
```

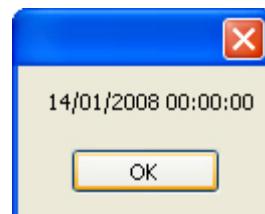
There are 12 different ways to use the DateTime structure. Each one is giving you the option of a date/time format. Examine option 4 of 12:

```
DateTime firstDate = new DateTime()
▲ 4 of 12 ▾ DateTime.DateTime (int year, int month, int day)
year: The year (1 through 9999).
```

You can just type some numbers here:

```
DateTime firstDate = new DateTime(2008, 01, 14);
```

What you’ll get back is the Date you specified. But you’ll also get the time as a series of zeros:



If you want to calculate the difference between one date and another, then a more precise structure is used with DateTime – TimeSpan. Here’s how to use it:

```
DateTime firstDate = new DateTime(2008, 01, 14);
DateTime secondDate = DateTime.Parse("1 Feb 2008");

TimeSpan dateDiff;
dateDiff = secondDate.Subtract(firstDate);
MessageBox.Show( "Date diff:" + dateDiff.ToString() );
```

The first line sets up a date: the 14th of January 2008. The second line sets up a date in a different way:

```
DateTime secondDate = DateTime.Parse("1 Feb 2008");
```

After the dot of DateTime, we’re using **Parse**. In between the round brackets of Parse, simply type your date between double quotes.

When you subtract one date from another, the answer is returned as a `TimeSpan` structure. This uses similar methods and properties as `DateTime`. One of the options is **Subtract**:

```
TimeSpan dateDiff;  
dateDiff = secondDate.Subtract( firstDate );
```

After typing a date then a dot, use the `Subtract` method. In between the round brackets of `Subtract`, you need another date – the one you’re trying to subtract. When the code is run, it will calculate the difference between the two dates.

You can also add date and time values to ones you already have. Examine this code:

```
firstDate = firstDate.AddDays(30);  
MessageBox.Show( firstDate.ToString("D") );
```

Now we’re using **AddDays** after our `firstDate` variable. In between the round brackets of `AddDays`, type how many days you want to add.

When the IntelliSense list appears, have a look at other date and time values you can add: `AddMonths`, `AddYears`, etc.

Play around with Date and Time values until you get the hang of them – they’ll come in handy one day!

Graphics

C# .NET allows you to draw straight to a form or form objects. You do all the drawing with inbuilt Graphic objects.

The first thing to do is to set up a Graphics object. You then specify what it is you want to draw on. To draw directly to a form, you can do this:

```
Graphics surface;
surface = this.CreateGraphics();
```

The first line sets up a Graphics variable that we've called **surface**. To set the current form as the graphics object that will be drawn to, use the keyword **this** followed by **CreateGraphics()**. If you wanted to draw on, say, a label instead, the code would be this:

```
surface = this.label_Name.CreateGraphics();
```

Once you have told C# which object you want to draw on you can then use a series of inbuilt tools to do the actual drawing. Let's draw a line.

C# uses Pens to draw shapes. To set up a new pen object, you need the colour the pen will use, followed by how wide you want the lines:

```
Pen pen1 = new Pen(Color.Blue, 1.0f);
```

Here, we've set up a new pen object called **pen1**. In between the round brackets of Pen, we have "**Color.Blue**". Color is an inbuilt class. After you type the dot, you should see the IntelliSense box appear with a list of colours. Simply select the one you want. After a comma, type the width of the pen. Our width was 1.0. We've added the "f" because you need a floating point number.

Once you have a pen tool, you can use **Draw** methods to draw to your object. In the code below, we're using the DrawLine method:

```
surface.DrawLine(pen1, 10, 10, 100, 100);
```

We're using the DrawLine method to draw on our surface object. In between the round brackets of DrawLine, you first need the name of your Pen. The numbers are coordinates. The first two (10, 10) mean 10 units from the left and 10 units down. This will be the starting point of the line. The second two (100, 100) are the end point of the line. So 100 units from the left and 100 units down.

To test it out, create a new project. Click your form to select it. Now have a look at the properties box on your right. Click the lightning bolt to see a list of events that the form has. Locate the **Click** event. Double click the word Click to bring up the code for that event. Now type the following code:

```
private void Form1_Click(object sender, EventArgs e)
{
    Graphics surface = this.CreateGraphics();

    Pen pen1 = new Pen(Color.Blue, 1.0f);

    surface.DrawLine(pen1, 10, 10, 100, 100);
}
```

Run your programme and click the form. You should find that a diagonal line is drawn, starting from the top left.

However, there is a problem with drawing straight to a form or form object. If you minimize the form and then maximize it, you'll find that all of the things you drew are erased. To prevent this, you can use the form **Paint** event. Let's see how.

Stop your programme and return to Design view. Click the form to select it. In the properties box on the right, click the lightning bolt to see the list again. Locate **Paint**, and then double click it (the word Paint). This will bring up the code stub for the form Paint event. Move your code from the Click event to the Paint event:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics surface = this.CreateGraphics();

    Pen pen1 = new Pen(System.Drawing.Color.Blue, 1.0f);

    surface.DrawLine(pen1, 10, 10, 100, 100);
}
```

Run your programme to see the diagonal line again. When you Minimize then Maximize the form, this time the line is not erased.

The Rectangle Class

Another useful graphics class is Rectangle. As its name suggests, it's used to draw rectangles. But, strangely, it can also be used to draw circles and ellipses.

Add the following to your code, just beneath the third line:

```
Rectangle recOne = new Rectangle(150, 10, 50, 200);
```

This sets up a new Rectangle object called **recOne**. In between the round brackets of Rectangle, we have two sets of numbers. The first set (150, 10) is how far to the left you want your rectangle, and how far down. The second set (50, 200) is the width of the rectangle, and then the height.

To draw the rectangle, the DrawRectangle method is used:

```
surface.DrawRectangle( pen1, recOne );
```

We're drawing to our Graphics object, the one we've called **surface**. In between the round brackets of DrawRectangle, we first have the pen we want to use, followed by the rectangle object.

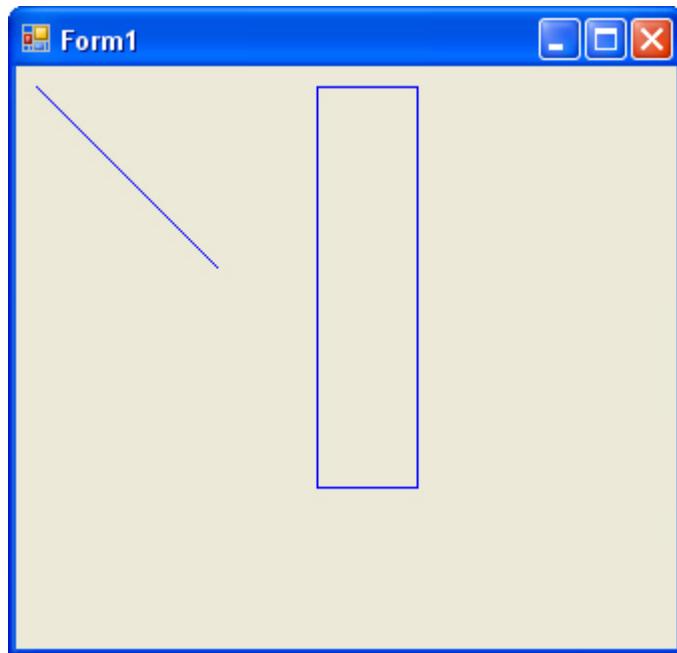
Add the new line to your code. Your coding window will then look like this:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics surface = this.CreateGraphics();
    Pen pen1 = new Pen(System.Drawing.Color.Blue, 1.0f);
    surface.DrawLine(pen1, 10, 10, 100, 100);

    Rectangle recOne = new Rectangle(150, 10, 50, 200);

    surface.DrawRectangle(pen1, recOne);
}
```

When you run your form, you should see something like this:



You can also specify rectangle values separately. Examine this new code:

```
Rectangle recTwo = new Rectangle();

recTwo.X = 20;
recTwo.Y = 20;
recTwo.Width = 100;
recTwo.Height = 100;

surface.DrawRectangle(pen1, recTwo);
```

This time, the round brackets of Rectangle have been left empty. The position of the rectangle is done with the X value (how far to the left) and the Y value (how far down). Width and Height are then on separate lines. The rectangle is drawn to the Graphic object (surface, for us) in the same way.

If you wanted to draw a circle or ellipse, it is drawn inside of a rectangle. The only thing you need to do is to use the DrawEllipse method:

```
surface.DrawEllipse(pen1, recOne);
```

If you want, you can set up some new position and size values for your rectangle, all within the round brackets of DrawEllipse:

```
surface.DrawEllipse(pen1, 100, 10, 200, 200);
```

The first two numbers are the X and Y coordinates. The second two numbers are the dimensions of the Ellipse. 200, 200 will get you a circle.

Brushes

If you want a filled rectangle instead of a blank one, you need a SolidBrush object. The Brush can then be used to Fill shapes.

Add the following to your code:

```
SolidBrush brushOne = new SolidBrush(Color.Blue);
```

So we've set up a new SolidBrush and called it **brushOne**. In between the round brackets of SolidBrush, we've specified a colour.

To use your brush, you use the Fill method of the graphic object you set up:

```
surface.FillRectangle( brushOne, recTwo );
```

To fill our rectangle, we're using the **FillRectanlge** method. In between the round brackets of FillRectanlge you need the name of a brush, followed by the rectangle that you want to fill.

Add the new line to your code. When your form is run, you should see that your rectangle has a solid fill.

As well as a solid fill, you can have other types of fill. To do so, you need a different type of brush. There a few to choose from:

HatchBrush
LinearGradientBrush
PathGradientBrush
TextureBrush

In the code below, we're using a HatchBrush instead of a SolidBrush:

```
HatchBrush brushTwo;
brushTwo = new HatchBrush(HatchStyle.DiagonalCross,
                           Color.Blue, Color.AliceBlue);
```

In order to get the HatchBrush, you need to add a new **using** line at the top:

```
using System.Drawing.Drawing2D;
```

In between the round brackets of HatchBrush, you select a HatchStyle from the IntelliSense list. After that, you generally need two colours. We've chosen blue and AliceBlue.

Drawing Polygons

You can also draw irregular shapes, like Polygons. To do that, you set up a series of points, and hand them to the DrawPolygon method.

The first thing to do is to set up an array of points:

```
Point[] polygonPoints = new Point[5];
```

This sets up a new point array called **polygonPoints**. We're specified 5 sets of points. The points are X, Y coordinates. X is how far to the left you want the point and Y is how far down. You set them up like this:

```
polygonPoints[0] = new Point(113, 283);
polygonPoints[1] = new Point(70, 156);
polygonPoints[2] = new Point(180, 70);
polygonPoints[3] = new Point(290, 156);
polygonPoints[4] = new Point(250, 283);
```

So each slot in the array is filled with a **new** point. In between the round brackets of Point, you add your X, Y coordinates.

Once you have your points array, you can hand it to the DrawPolygon method:

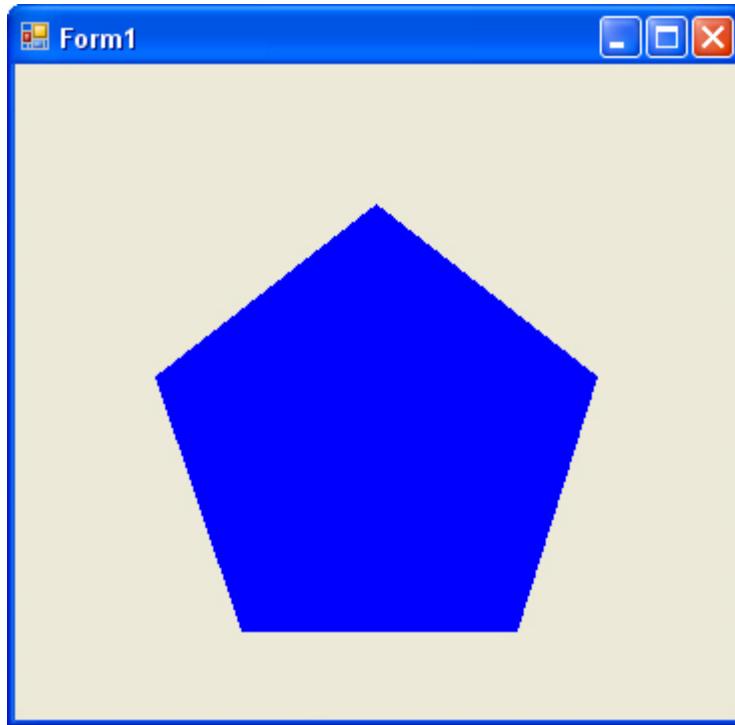
```
surface.DrawPolygon(pen1, polygonPoints);
```

After specifying what you want to draw on (**surface**, for us), you type a dot followed by **DrawPolygon**. In between the round brackets of DrawPolygon you first need a pen. Then you need your points array. Note that the brackets of the array are not needed.

If you want to fill your polygon, you can use the FillPolygon method:

```
surface.FillPolygon(brushOne, polygonPoints);
```

FillPolygon needs a brush, and the points array again. When the above code is run, it will produce the following polygon:



Drawing Text

You can also draw text to a form. You set up a font, a brush, an optional text direction, and then draw your text to the form. Here's an example:

```
Font drawFont = new Font("Arial", 20);
SolidBrush drawBrush = new SolidBrush(Color.Black);

surface.DrawString("Some Text", drawFont, drawBrush, 50.0f, 20.0f);
```

So you set up a Font object and specify the font and font size in between the round brackets of Font. We've created a new brush for this, but you can use a brush that you already have. To draw the string to the form, you use **DrawString**. In between the round brackets of DrawString, you have a lot of different options. In ours, we first have the text we want to draw. Then we have the Font and the brush. The numbers are where you want to draw the text. We've specified 50 across and 20 down. The values are floating point values.

If you want vertical text, you have to set up something called a StringFormat. Like this:

```
StringFormat drawFormat = new StringFormat();
drawFormat.FormatFlags = StringFormatFlags.DirectionVertical;
```

After the dot of StringFormatFlags, you'll see a list of options. Vertical is one is these. You then use your StringFormat in between the round brackets of DrawString:

```
Font drawFont = new Font("Arial", 20);
SolidBrush drawBrush = new SolidBrush(Color.Black);

StringFormat drawFormat = new StringFormat();
drawFormat.FormatFlags = StringFormatFlags.DirectionVertical;

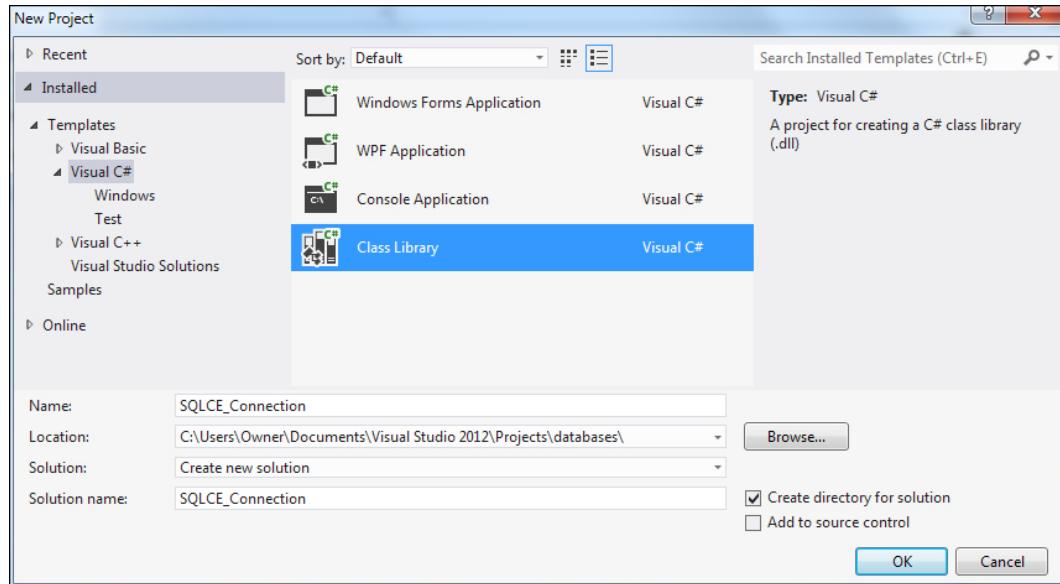
surface.DrawString("Text", drawFont, drawBrush, 50.0f, 20.0f, drawFormat);
```

There's obviously a lot more to you can do with Graphic objects. But we hope we've given you a taster!

How to Create a Class Library File

You can create a class library of your code, save it as a DLL and then reuse your C# code in other projects. The process is quite straightforward.

Start a new project. From the New Project dialogue box, select **Class Library** instead of Windows Forms Application. In the Name box, type a name for your Class Library. Call it **SQLCE_Connection** for this lesson. Then click OK:



When you click OK, C# will create a Namespace with the Name you've just used, and you'll be looking at the code window:

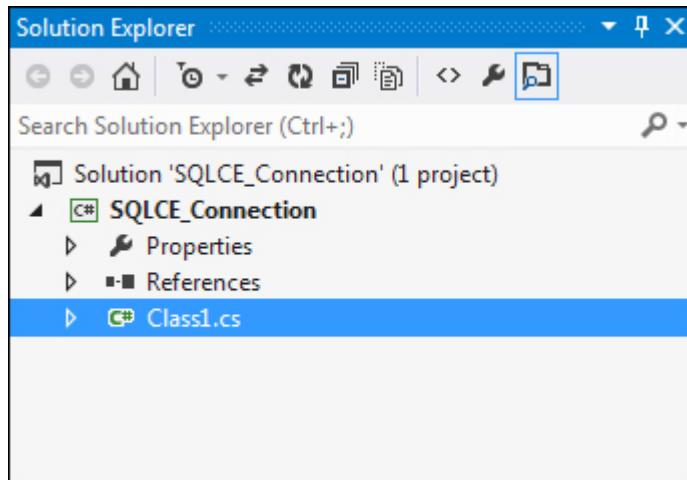
```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

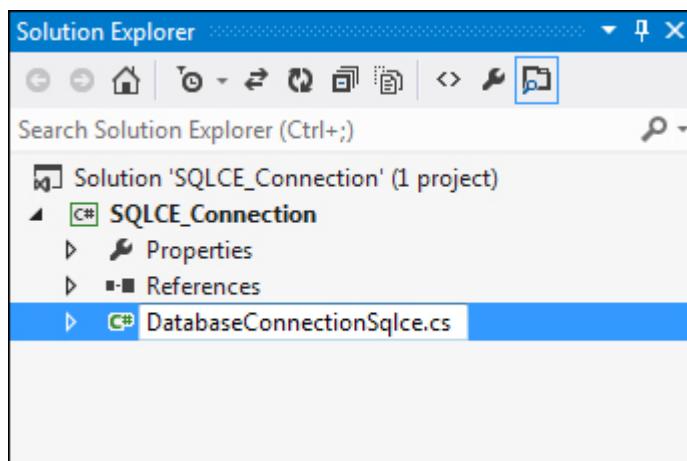
namespace SQLCE_Connection
{
    public class Class1
    {
    }
}

```

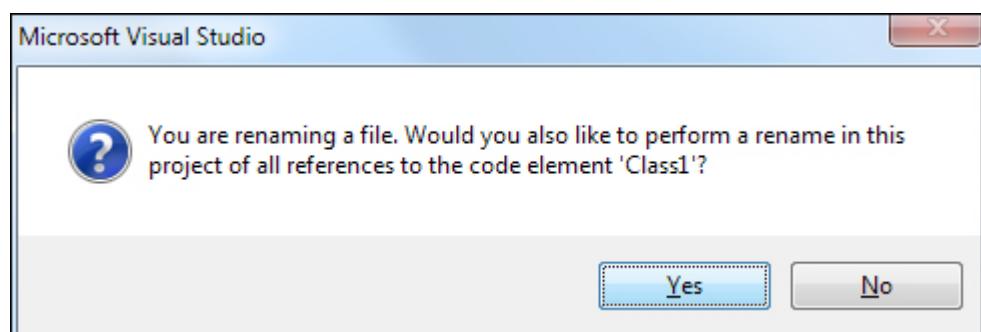
Note that the class is called **Class1**. You can change that to anything you like. But we'll change it to **DatabaseConnectionSqlce**. So locate **Class1.cs** in the Solution Explorer on the right:



Right click, and select **Rename** from the menu. Rename it to **DatabaseConnectionSqlce.cs**:



Press the Enter key on your keyboard, and you will then get a message box telling you the following:



Say **Yes** to the message, and your code window will then look like this:

Class Library Files

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SQLCE_Connection
{
    public class DatabaseConnectionSqlce
    {
    }
}
```

Class1 has now been renamed to something more useful. The class we're going to create, as its name suggests, will help us to connect to a SqlCe database. We'll add fields, properties and a method to the class. Once the class is created, we'll reuse it in another project.

Here's the whole code for the class: (We'll go through it in a moment.)

```
namespace SQLCE_Connection
{
    public class DatabaseConnectionSqlce
    {
        private string sql_string;
        private string strCon;

        public string Sql
        {
            set { sql_string = value; }
        }

        public string connection_string
        {
            set { strCon = value; }
        }

        public System.Data.DataSet GetConnection
        {
            get
            { return MyDataSet(); }
        }

        private System.Data.DataSet MyDataSet()
        {
            System.Data.SqlClient.SqlCeConnection con = new
            System.Data.SqlClient.SqlCeConnection(strCon);

            con.Open();

            System.Data.SqlClient.SqlCeDataAdapter da_1 = new
            System.Data.SqlClient.SqlCeDataAdapter(sql_string, con);
            System.Data.DataSet dat_set = new System.Data.DataSet();
            da_1.Fill(dat_set, "test");
            con.Close();

            return dat_set;
        }
    }
}
```

Before adding it yourself, if you have Visual Studio Express 2012, you'll need to add a reference to SqlCe like you did before (**Project > Add Reference >Extensions**, then scroll down and locate **System.Data.SqlServerCe**. Put a check in the box and click OK.)

We'll now go through the code for the class.

The first thing we do is to set up to field variables:

```
private string sql_string;
private string strCon;
```

The **sql_string** variable will hold a SQL string like "SELECT * FROM table". The **strCon** variable will hold a location of the database.

The next code is a write-only property:

```
public string Sql
{
    set { sql_string = value; }
}
```

We've called this property **Sql**. The **value** (the SQL itself) will be assigned to the **sql_string** field variable.

The next bit of code sets up a connection string property. (You saw how to set up a connection string in the database section.) Again, this is a write-only property (meaning it has no **get** part).

```
public string connection_string
{
    set { strCon = value; }
}
```

The third property returns a Dataset. It's this:

```
public System.Data.DataSet GetConnection
{
    get
    { return MyDataSet(); }
}
```

We've called this property **GetConnection**. This is a read-only property. For the **get** part we're calling a method with the name **MyDataSet**. The method connects to the database and fills a Dataset.

The final part of the class is the method from above – **MyDataSet**. The code is this:

```

private System.Data.DataSet MyDataSet()
{
    System.Data.SqlClient.SqlCeConnection con = new
    System.Data.SqlClient.SqlCeConnection(strCon);

    con.Open();

    System.Data.SqlClient.SqlCeDataAdapter da_1 = new

    System.Data.SqlClient.SqlCeDataAdapter(sql_string, con);
    System.Data.DataSet dat_set = new System.Data.DataSet();
    da_1.Fill(dat_set, "Table_Data_1");
    con.Close();

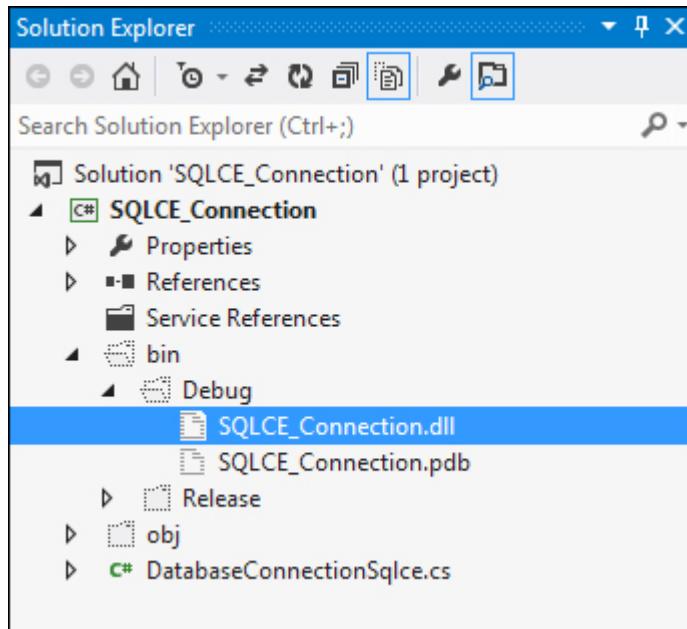
    return dat_set;
}

```

This is code that you've met before. The method type is **DataSet**, rather than something like **int**, **string** or **bool**. That's why the return line at the bottom is a **DataSet** object. But we're just opening a connection to a database table using the connection string and SQL that we got from the properties.

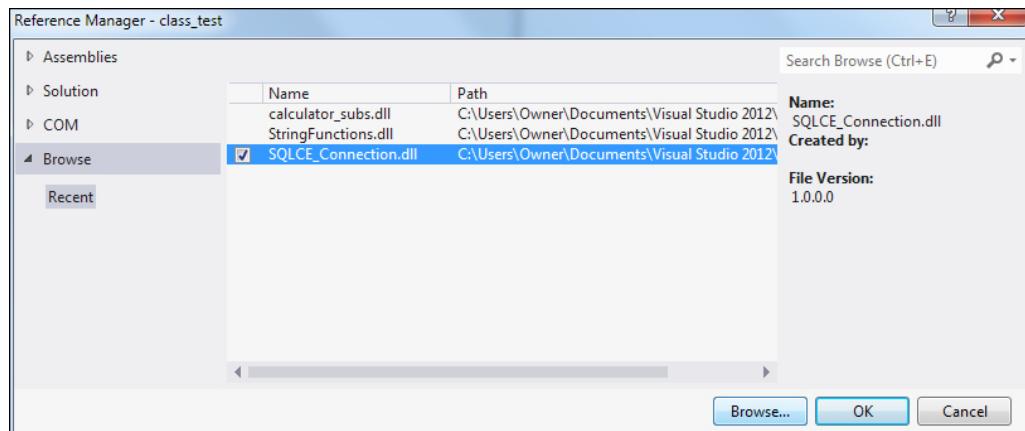
Now that we have a class all set up we can build it.

Click **Build > Build Solution** from the C# menu bars at the top. Click **File > Save All**, as well. Now have a look at the Solution Explorer again. Click the icon for All Files, and then expand the **bin > Debug** folder (expand the **Release** folder in C# 2010):



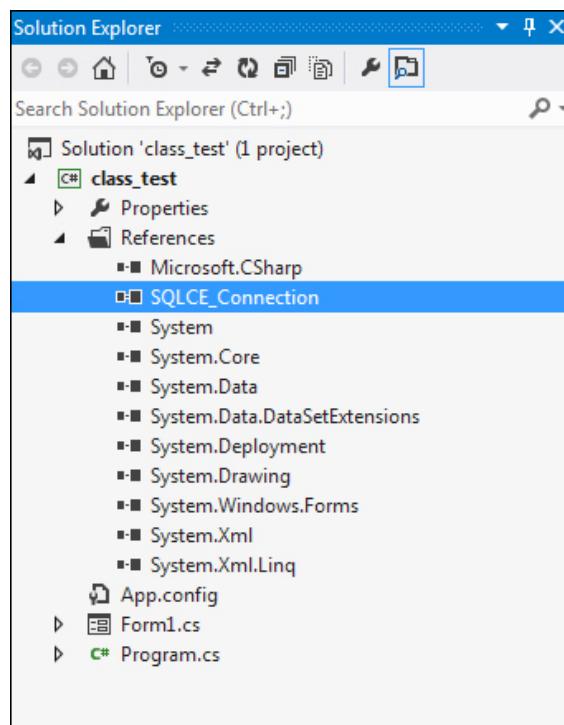
And there's our DLL file! We can use this in other projects. To test it out, click **File > Close Solution**. Now start a new Windows Forms Application project.

Add a new button to your form, and double click to get at the code. From the menu bars at the top of C#, click **Project > Add Reference**. You should see a dialogue box appear with a few tabs on it. Click the Browser tab at the top in versions prior to 2012. Or the Browser tab on the left in Visual Studio 2012:



Browse to where you saved your Class Library. Then locate the **Bin > Debug** folder from above (**Bin > Release** in 2010). Your DLL will be in there.

Select the **SQLCE_Connection** file and click OK to add the reference to your DLL. Have a look at the Solution Explorer on the right, and you should see it appear on the list of references:

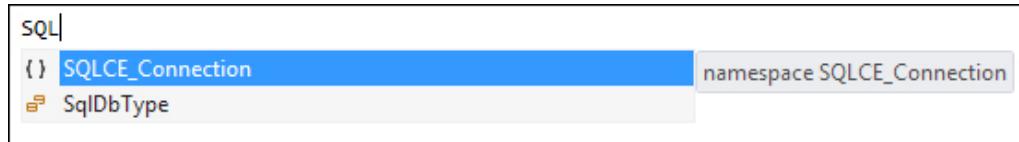


Now that you have a reference, you can use your code.

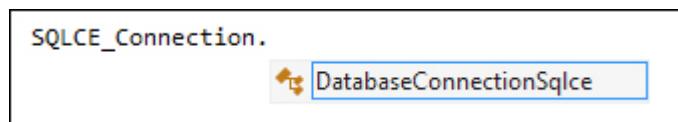
Add the following rather long line to your button code:

```
SQLCE_Connection.DatabaseConnectionSqlce objConnect = new  
SQLCE_Connection.DatabaseConnectionSqlce();
```

As soon as you type the namespace, **SQLCE_Connection**, you should see it appear on the Intellisense list:



After **SQLCE_Connection**, type a dot. You'll then see an option to add the Class name from the Intellisense list:



We're calling our new **DatabaseConnectionSqlce** object **objConnect**. After an equal sign, we create a new object from the class:

```
new SQLCE_Connection.DatabaseConnectionSqlce();
```

Now that we have an object set up from our class we can go ahead and use the its properties.

For the connection string, we can use the same one you saw in the database section, the one that connects to the Employees database:

```
"Data Source=C:\\Users\\Owner\\Documents\\Employees.sdf";
```

The SQL is the same as before, too:

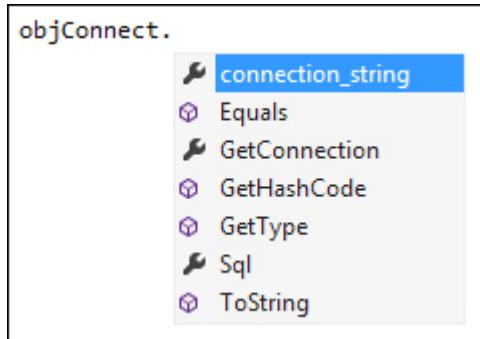
```
"SELECT * From tbl_employees";
```

So we want to connect to the table called **tbl_employees**.

To access the connection property from your Class, add the following (you can keep yours all on one line):

```
objConnect.connection_string = "Data  
Source=C:\\Users\\Owner\\Documents\\Employees.sdf";
```

As soon as you type a dot after **objConnect**, you should see the Intellisense list appear again:



And there are our three properties: **connection_string**, **GetConnection**, and **Sql**. Select the first one, **connection_string**, and then add the location to your database.

To add the SQL, the line is this:

```
objConnect.Sql = "SELECT * From tbl_employees";
```

To create a DataSet and get all the records back from the database, add this line:

```
DataSet ds = objConnect.GetConnection;
```

This sets up a DataSet object that we've called **ds**. Our read-only property, **GetConnection**, goes on the right of the equal sign.

To test if everything is OK, you can add the following lines (add two text boxes to your form, first):

```
DataRow dRow = ds.Tables[0].Rows[0];  
textBox1.Text = dRow.ItemArray.GetValue(1).ToString();  
textBox2.Text = dRow.ItemArray.GetValue(2).ToString();
```

This sets up a DataRow object and gets the first row from our **ds** object. If you get any errors when testing it out, make sure you have typed the name of your database table correctly in the SQL line, and that you have a database in the correct location.

You could, of course, add some error checking with a **try ... catch** block. The whole code would then be this:

Class Library Files

```
private void button1_Click(object sender, EventArgs e)
{
    try
    {
        SQLCE_Connection.DatabaseConnectionSqlce objConnect =
            new SQLCE_Connection.DatabaseConnectionSqlce();

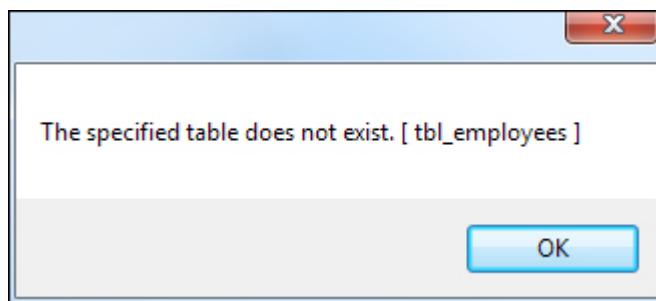
        objConnect.connection_string = "Data Source=C:\\\\Users\\\\Owner\\\\Documents\\\\Employees.sdf";
        objConnect.Sql = "SELECT * From tbl_employees";

        DataSet ds = objConnect.GetConnection;

        DataRow dRow = ds.Tables[0].Rows[0];

        textBox1.Text = dRow.ItemArray.GetValue(1).ToString();
        textBox2.Text = dRow.ItemArray.GetValue(2).ToString();
    }
    catch (Exception err)
    {
        MessageBox.Show(err.Message);
    }
}
```

The **try ... catch** block could then pick up any error. For example, here's an error message generated when the table name is incorrect:



But there you have it – an easy way to create your own class libraries and reuse your C# code.

C# Charts

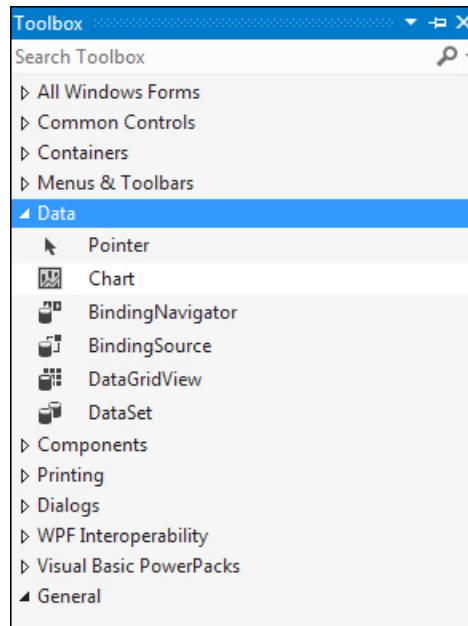
In this section, we'll take a look at how to create charts in C# .NET. We'll start with a column chart. By the end of this section, you'll have created something that looks like this:



Let's make a start.

Adding a Chart to a Windows Form

Start a new project for this and call it **Column_Chart**. Make your new form a little bit bigger. Now locate the chart control in the toolbox, in the **Data** category:



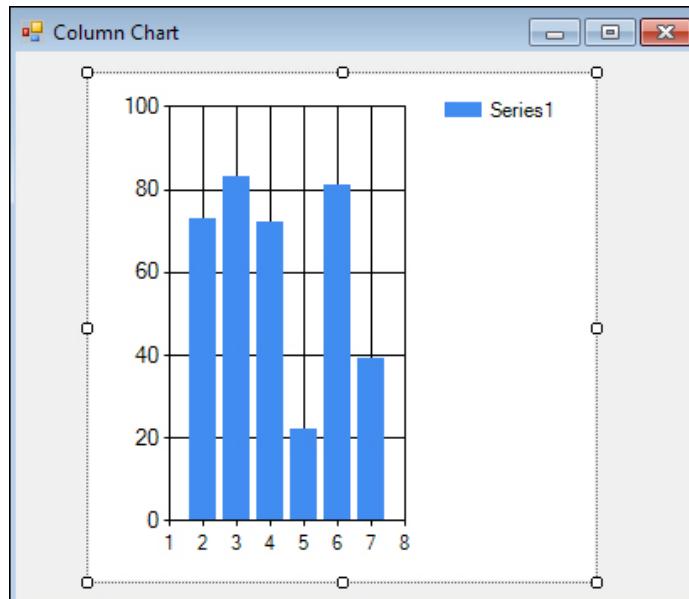
If you have version 2008 of C# .NET then you may need to download the chart control before you start. You can download the chart control here:

<http://www.microsoft.com/en-us/download/details.aspx?id=23903>

If you're still having problems after trying to install the control then check this page out:

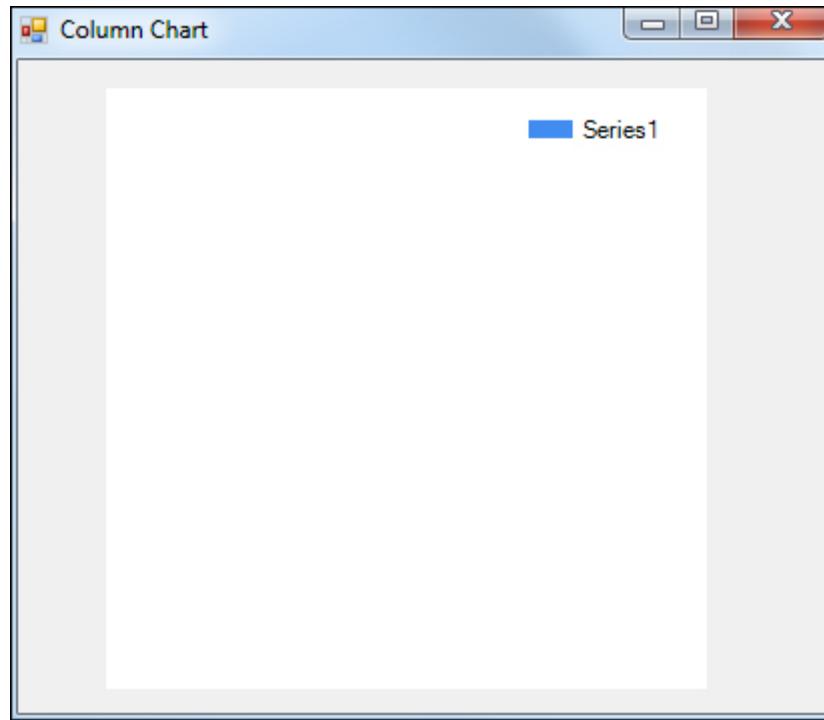
<http://msdn.microsoft.com/en-us/magazine/dd569763.aspx>

Once you've located the Chart Control, either drag and drop one onto your form, or simply double click it. Your form should look like this:

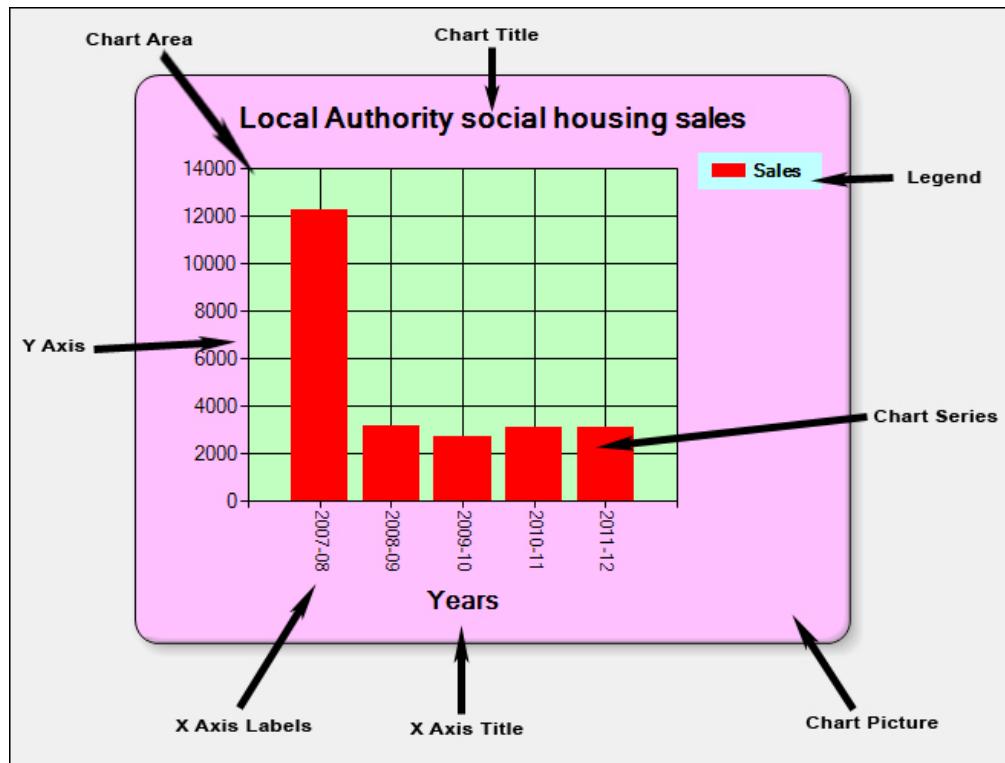


The default chart looks as though it already has column data. The Y Axis has numbers from 0 to 100 and the X Axis has the numbers 1 to 8. There is also a

Legend in the top right. However, Run your programme and you'll see that this column data disappears:



We'll add some column data using code. Before doing so, if you're not too familiar with chart terminology, here's an image to study:



And here's what it all means:

Chart Picture

This is the total area that a chart occupies, the pink parts and chart border in the image. A Chart Picture can actually be an image, if you want, but it's more usually just a colour.

Chart Area

A chart inside of a Chart Picture. Chart Areas are collections, which means that you can have more than one chart area inside of your Chart Picture.

Chart Title

The Title for the chart, which appears at the top of the Chart Area.

Chart Series

This is the data that goes to make up your chart. A Chart Series is a collection of Data Points.

Legend

A chart legend tells you information about the data in the chart. It also contains the colour of a Chart Series.

X Axis and Y Axis

The X Axis runs from left to right, at the bottom of the chart. The Y Axis runs from the bottom left to the top left.

X Axis and Y Axis Title

You can add a title to the X and Y Axis, so as to explain what they mean, and what the data relates to.

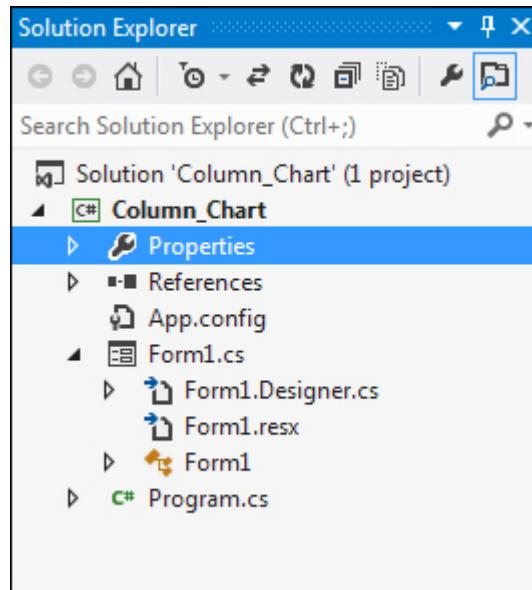
Axes Labels

You'll want to add some labels to the Axes, so as to explain what the data means. In our example above, we have a series of years for the X Axis Labels. The Y Axis Labels are numbers automatically added by C#.

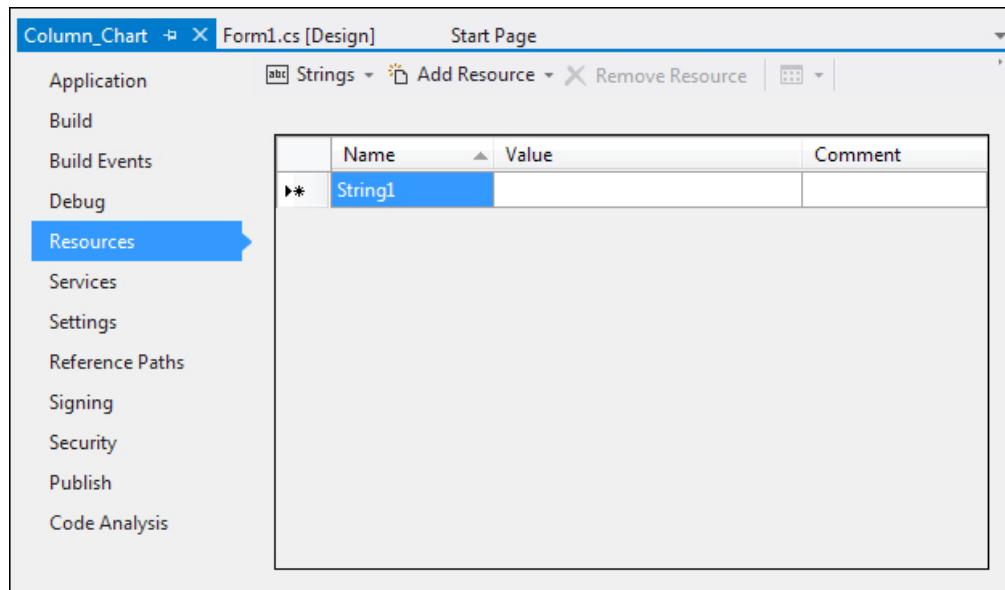
Adding the Data for a Chart

The data in our chart relates to the sale of Local Authority housing in the UK. It shows how many houses each year a local authority (a council) sold to its tenants. What we'll do is to create a text file with this data in it, and then add the text file as a project resource.

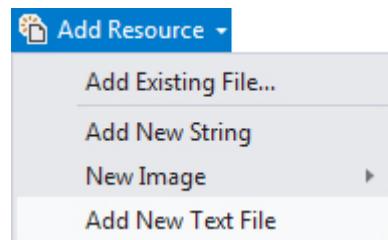
So locate the **Properties** item in the solution Explorer on the right:



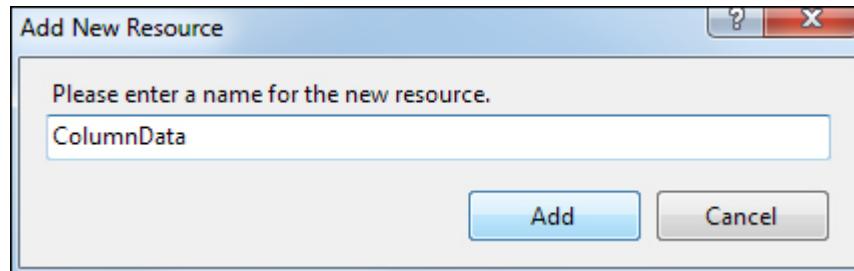
Double click the Properties item and you'll see a new tab open up. Click on Resources from the list on the left of the new tab:



Now click the Add Resource button at the top. From the list of options, select **Add New Text File**:



When you click **Add New Text File** you'll be asked to give the new file a name:



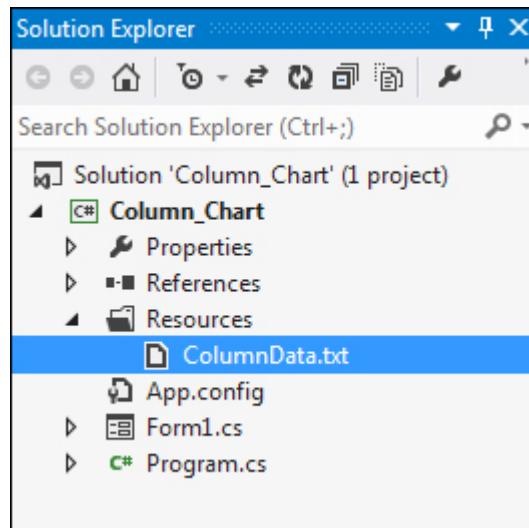
Type **ColumnData** and then click the Add button.

When you click **Add**, you'll see a blank text file upon up as a new tab. Type the following numbers (don't hit the enter key on your keyboard at the end of the line):

12250, 3150, 2730, 3090, 3080

When you've typed the numbers, save your work so far, and close down the text file. You can also close the properties tab, as well.

Now have a look at the Solution Explorer on the right. You should see a Resources item, with you text file in it:



(Incidentally, you could add images to your resources folder, if you wanted to. You do it in the same way: double-click Properties, then select the **Resources** item on the left. From the **Add Resources** button, select **Add Existing Item**. Search for an image file and it will end up in your Resources folder, ready to be used in your project.)

Now that we have some data for our chart, though, it's time to do some coding.

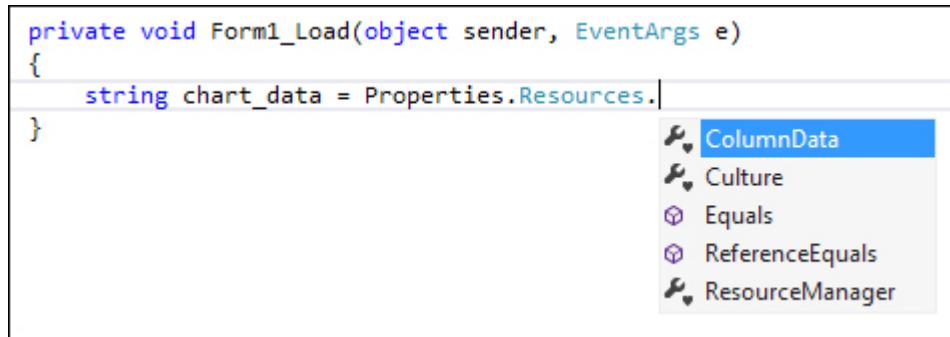
Adding Data to a Chart at Runtime

To add some data to our chart, double click the form (not the chart). This will open a Form Load event.

To get at the text file in the Resources folder, add the following line:

```
string test = Properties.Resources.ColumnData;
```

As soon as you type the dot after Resources, you should see the Intellisense menu appear:



Your text file, which was called `ColumnData`, should be on the list (without the extension `.txt`). What we're doing here is placing the entire contents of the text file into a string variable called `chart_data`.

However, we need all this data in an array because we're going to be looping round adding data points and Axis Labels. To turn your text file string into an array, simply use the inbuilt Split method:

```
string[] arrayData = chart_data.Split(',');
```

Each number in our text file was separated by a comma. In between the round brackets of Split, then, we have typed a comma in between single quotes. When the variable called `chart_data` is split, it will place all the pieces into an array we've called `arrayData`.

For the X Axis labels, we'll set up an array. But we need to set up some custom labels, as the X Axis labels are usually added automatically by C#. So add the following string array to your code:

```
string[] dates = new string[5] { "2007-08", "2008-09", "2009-10", "2010-11", "2011-12" };
```

Now add a CustomLabel object:

```
CustomLabel customLabel;
```

You may see a wavy line under `CustomLabel`. If so, then add the following `using` statement to the top of you code:

```
using System.Windows.Forms.DataVisualization.Charting;
```

The wavy underline should now disappear.

As well as a CustomLabel object we need an Axis object. We'll set this to the X Axis. Here's the line of code to add:

```
Axis axisX = chart1.ChartAreas[0].AxisX;
```

The chart Axes are part of the ChartArea. ChartAreas are part of a collection. To specify which ChartArea you want, you can type an index number between square brackets. We only have one ChartArea so the index is 0.

You can also refer to the ChartArea by name:

```
chart1.ChartAreas["ChartArea1"].AxisX;
```

When you add a chart to a form, the default name for the ChartArea is **ChartArea1**. You can type this name between double quotes instead of an index number.

But your code so far should look like this: (We've spread our **dates** array over three lines so as to fit on this page. You can keep yours on one line.)

```
private void Form1_Load(object sender, EventArgs e)
{
    string chart_data = Properties.Resources.ColumnData;
    string[] dates = new string[5] {
        "2007-08", "2008-09", "2009-10", "2010-11", "2011-12"
    };

    string[] arrayData = chart_data.Split(',');
    CustomLabel CustomLabel;
    Axis axisX = chart1.ChartAreas[0].AxisX;
}
```

Time for the loop. First, set up some loop variables:

```
int endLoop = arrayData.Length;
double axelLabelPos = 0.5;
```

Now add the for loop:

```
for (int i = 0; i < endLoop; i++) {

}
```

The loop goes from 0 to how many elements are in the array called **arrayData**. This value is stored in the **endLoop** variable.

What we're going to be doing in the loop is setting up some data points. If your chart is a column chart then a point is how high a particular column is. Our chart data is stored in the array called **arrayData**. We have five numbers which we'll turn into 5 data points, and thus have 5 columns in our chart.

One complication we have is that each value in **arrayData** is text. Data points, however, can't be text – they have to be numbers. So we need to convert each text number in the array to an integer

As the first line of the for loop code, then, add the following:

```
int x = Int32.Parse( arrayData[i] );
```

We're using **Int32.Parse** to convert the textual numbers into integers. In between the round brackets of **Parse** we have our array.

Now that we have a number we can add it as a point. Points are added to the Points collection which in turn is part of a Series collection. Here's the next line of the loop code:

```
this.chart1.Series["Series1"].Points.Add(x);
```

The keyword **this** means "this form". After a dot, you type the name of your chart. The default name when you drag and drop a chart control onto a form is **chart1**. We haven't changed this name. After another dot, we have the **Series** collection. Because it is a collection, you need the name of a specific collection between square brackets. The default name for a series collection is **Series1**. If you add another series it will be called **Series2**. Add a third series and it will be **Series3**, and so on. The name of the Series goes between double quotes. (We'll be changing the Series name shortly.)

Type another dot after the Series name and you can access the **Points** collection. The Points collection has an **Add** method. In between round brackets of **Add**, you type a number (or variable, which is **x** for us). The number (point) is how high your column will be.

The final thing we need to do in the loop is to add the label for the X Axis.

We already have a **CustomLabel** object set up, and an X Axis Object. We can use these to add the labels. Here's the code:

```
customLabel = axisX.CustomLabels.Add(axelLabelPos,  
axelLabelPos + 1, dates[i]);
```

The code is a bit long, and spread out over two lines in this book. (You should keep yours on one line.) First we have our **customLabel** variable. After an equal sign, we have our **axisX** variable. Type a dot and you'll see an option for the **CustomLabels** collection. Select this and type another dot. You'll then see an option for the **Add** method. The Add method needs three things: a start position for the label, and end position, and then the text you want for your label.

Our first start position was set at 0.5. The end position is 1.5. So we're really saying this:

```
customLabel = axisX.CustomLabels.Add(0.5, 1.5, "2007-08");
```

The next starting position will be 1.5, with the end position at 2.5:

```
customLabel = axisX.CustomLabels.Add(1.5, 2.5, "2008-09");
```

The third label would then be this:

```
customLabel = axisX.CustomLabels.Add(2.5, 3.5, "2007-08");
```

What the numbers do is to position the label text neatly under the data column.

The final line of the for loop moves the value of **axelLabelPos** variable on by 1.0:

```
axelLabelPos = axelLabelPos + 1.0;
```

The whole of your code should now look like this:

```
private void Form1_Load(object sender, EventArgs e)
{
    string chart_data = Properties.Resources.ColumnData;
    string[] dates = new string[5] {
        "2007-08", "2008-09", "2009-10", "2010-11", "2011-12"
    };

    string[] arrayData = chart_data.Split(',');
    CustomLabel customLabel;
    Axis axisX = chart1.ChartAreas[0].AxisX;

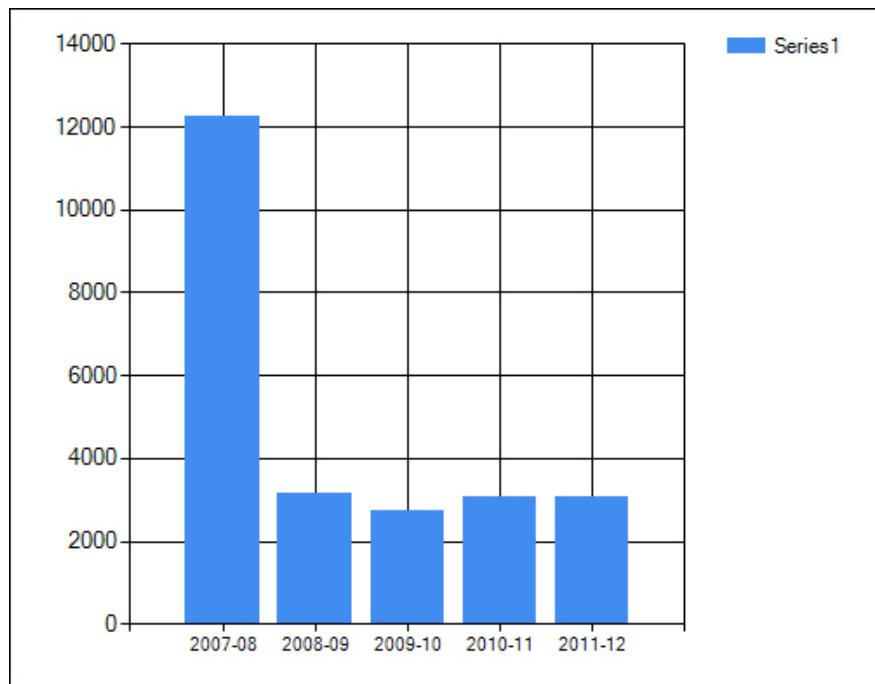
    int endLoop = arrayData.Length;
    double axelLabelPos = 0.5;

    for (int i = 0; i < endLoop; i++)
    {
        int x = Int32.Parse(arrayData[i]);

        this.chart1.Series["Series1"].Points.Add(x);

        customLabel = axisX.CustomLabels.Add(axelLabelPos, axelLabelPos + 1, dates[i]);
        axelLabelPos = axelLabelPos + 1.0;
    }
}
```

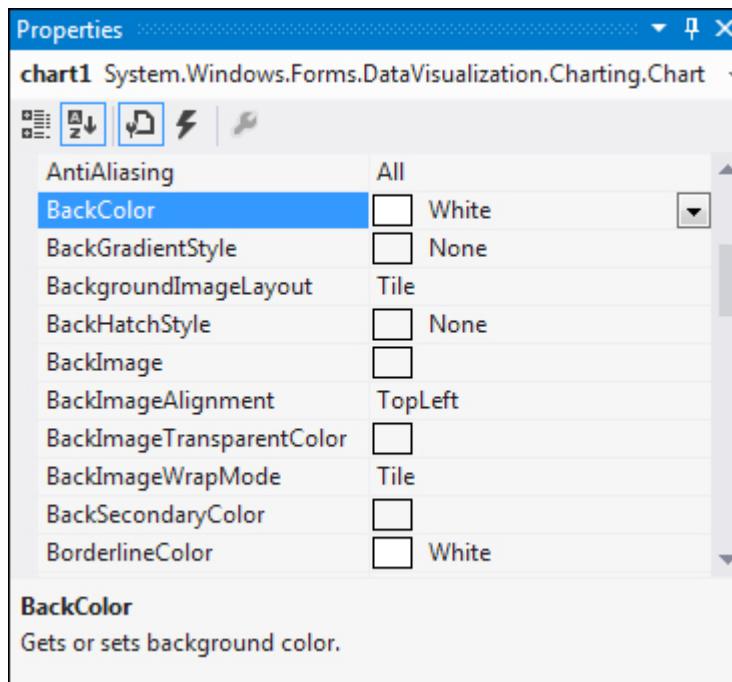
You can run your programme now to see what it all looks like. You should see something like this:



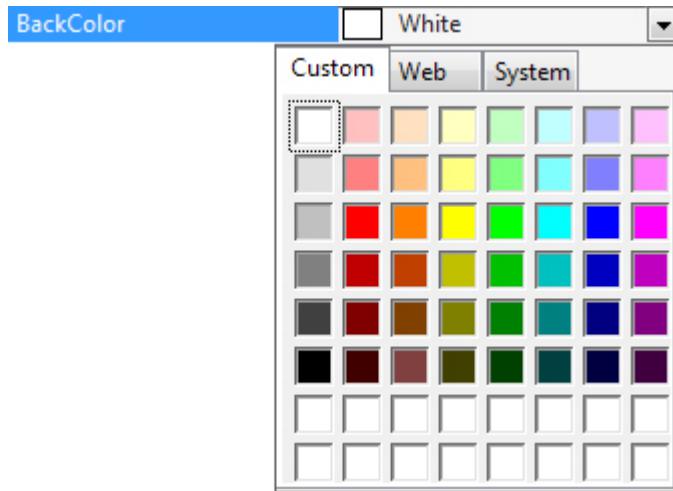
As you can see, the data from the text file is used for the column heights, while our data labels are on the bottom. The chart, however, looks a little bland. We'll now explore ways to enhance charts.

Chart Formatting

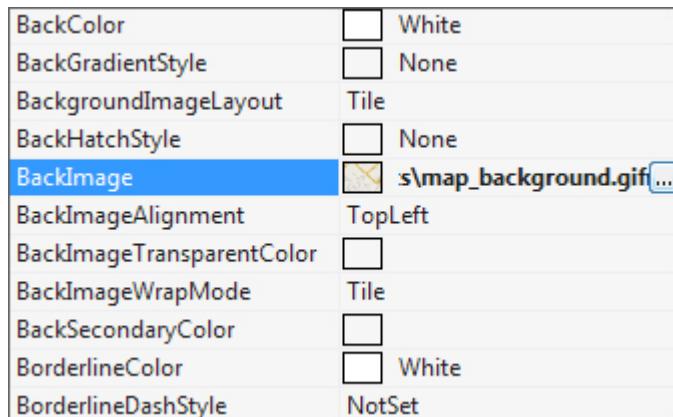
The first thing we can do is change the background colour of the chart. Click on your chart to select it. Now locate the **BackColor** property in the Properties area:



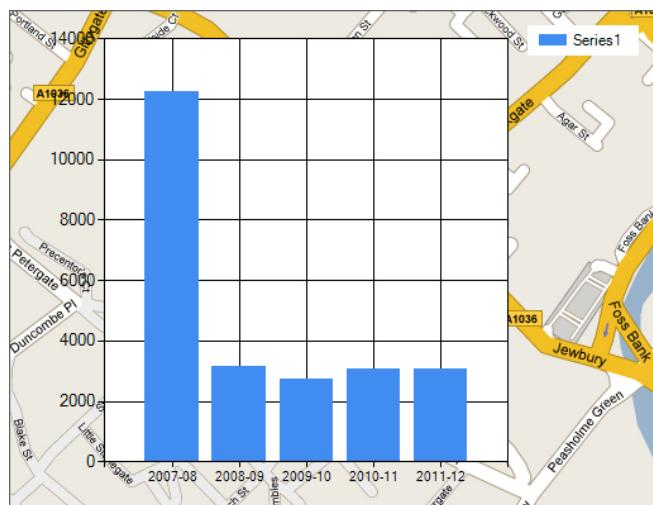
Click the arrow to reveal a dropdown box of colours you can choose from, and select the colour you want:



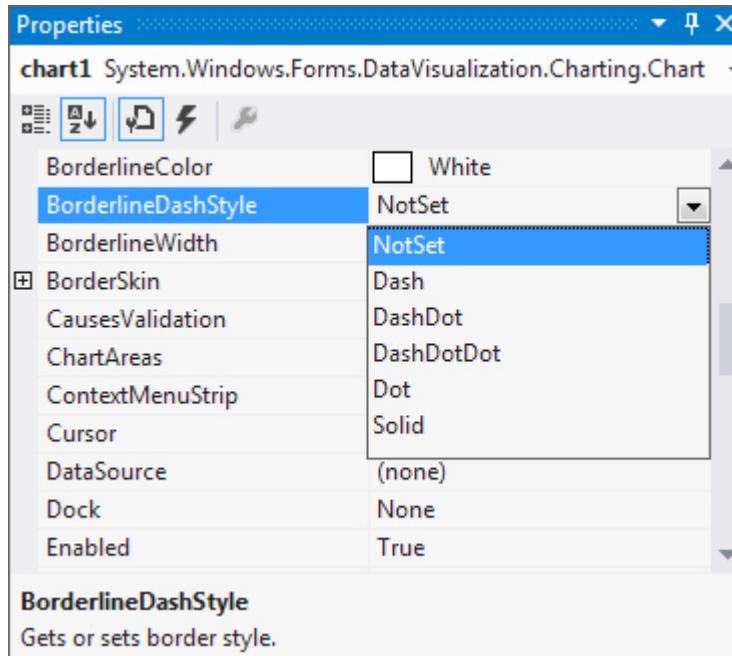
If you want an image rather than a colour for your background, click the **BackImage** property instead of **BackColor**. Click the button on the right, then navigate to where your image is stored:



In the screenshot below, we've chosen a map image as a background:



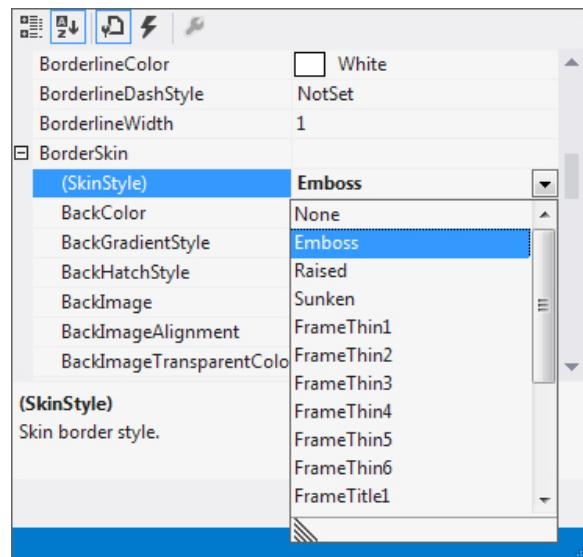
If you wanted a different border style for your chart the property to use is **BorderlineDashStyle**. Click the arrow to see a list of borders you can choose from:



Once you have selected a style, click the **BorderLineColor** property to set a new border colour.

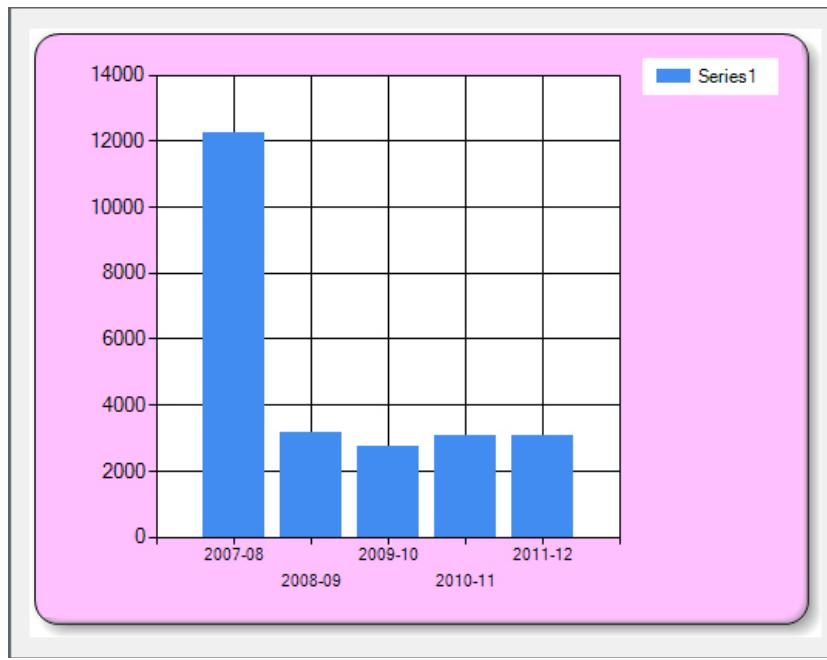
Try the various border styles and colours to see what they look like.

If you want a chart with rounded corners then click the **BorderSkin** property. Click the cross on the left to reveal more options. Click the arrow of **SkinStyle** to see the different skins you can apply to a chart:

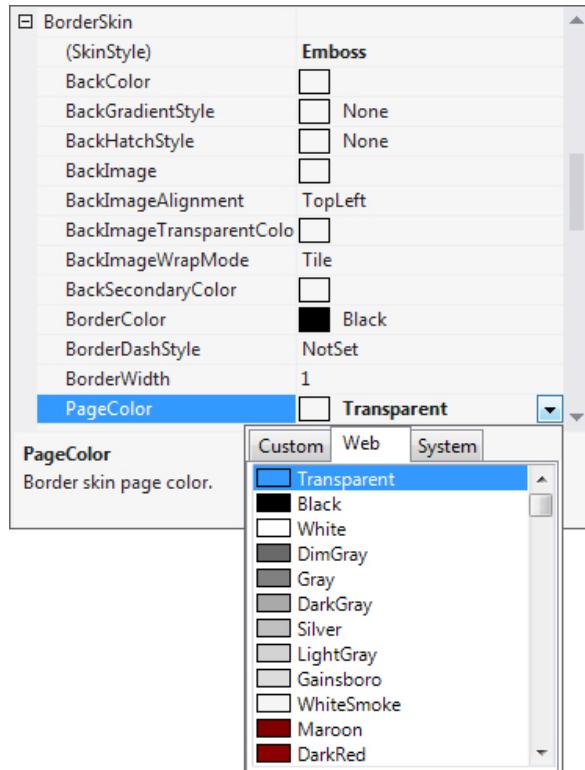


Select **Emboss** to get rounded corners.

Once you've made the above changes, your chart will look something like this:



One thing to notice here is the white background behind the chart, especially where the rounded corners are. To get rid of this, you can set this white background to be transparent. The property you need is **PageColor**, under the **BorderSkin** options:



Click the down arrow of **SkinColor** and select **Transparent**. The chart would then look like this:

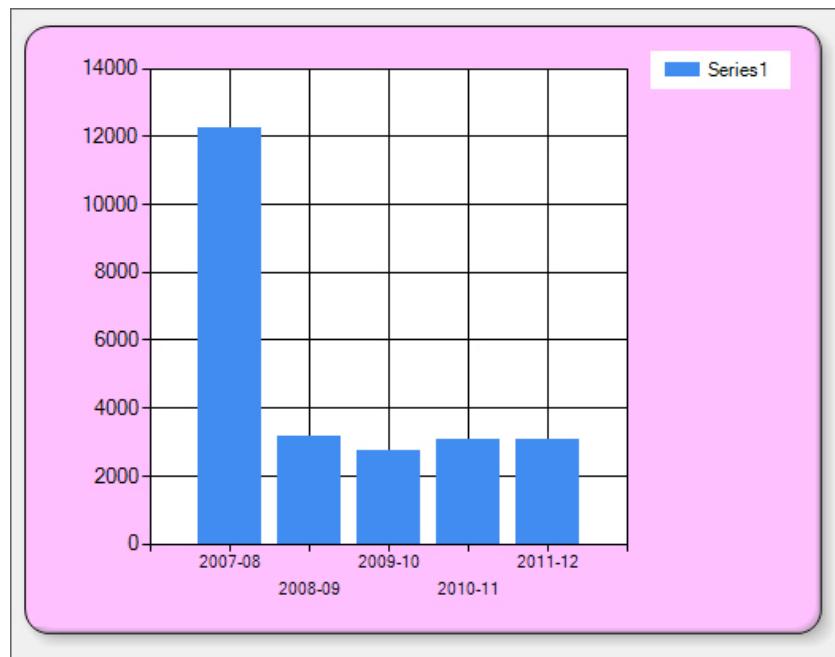
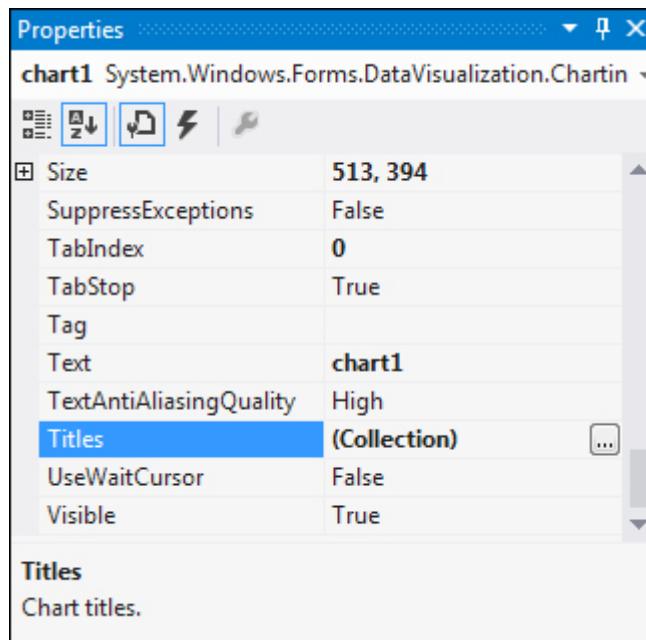
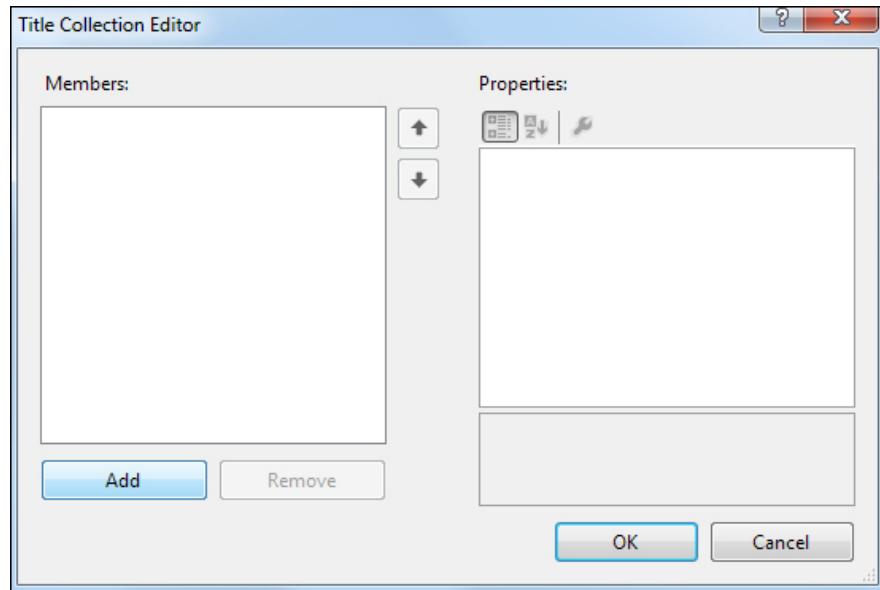


Chart Titles

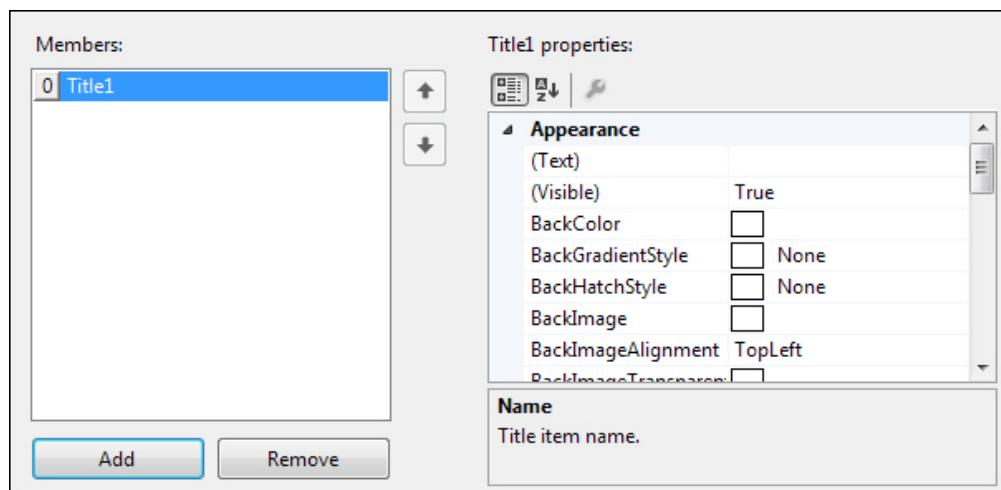
You can set a title at the top of your chart. To do this, you need to access the **Titles** collection. Click your chart to select it. In the properties area, locate the **Titles** item:



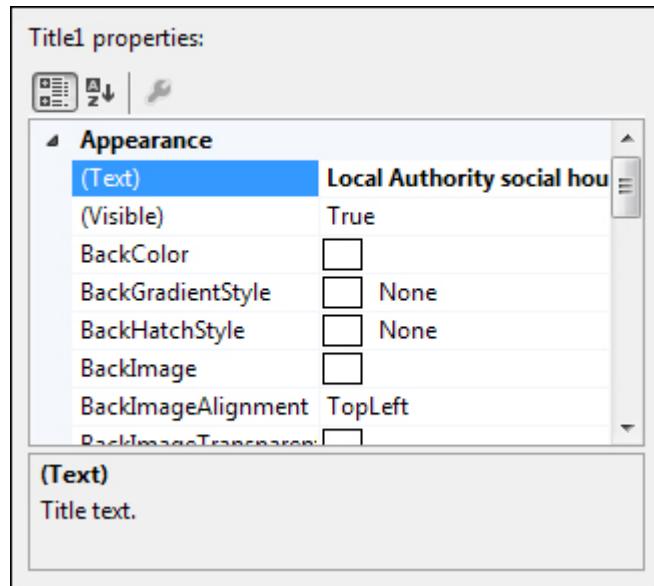
Click the button to the right of **Collection** to see the following dialogue box appear:



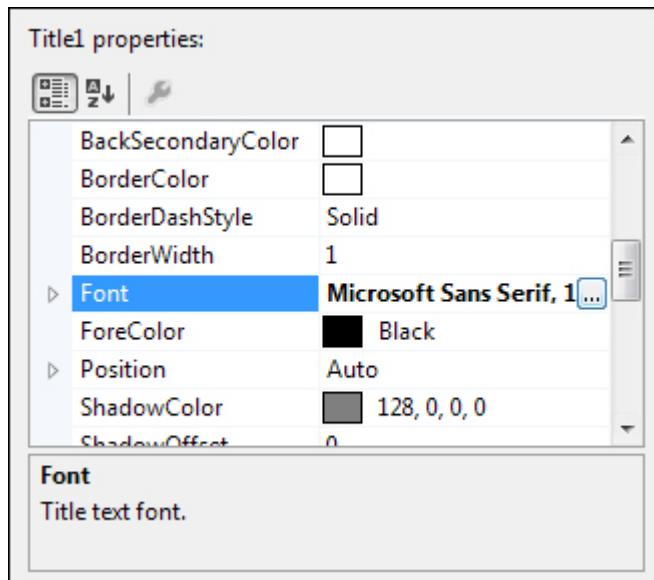
Click the Add button to add a new title. You'll then see a new item, **Title1**, appear in the Members area on the left:



Each title comes with its own properties. In the **Appearance** area of the title properties, type a new title:



To set the font style, scroll down and locate the Font property:



Click the button on the right of the Font area to set a font style for your title.

While you have the Titles dialogue box displayed, have a play around in the appearance category. See what the **BackColor** and **BorderColor** properties do.

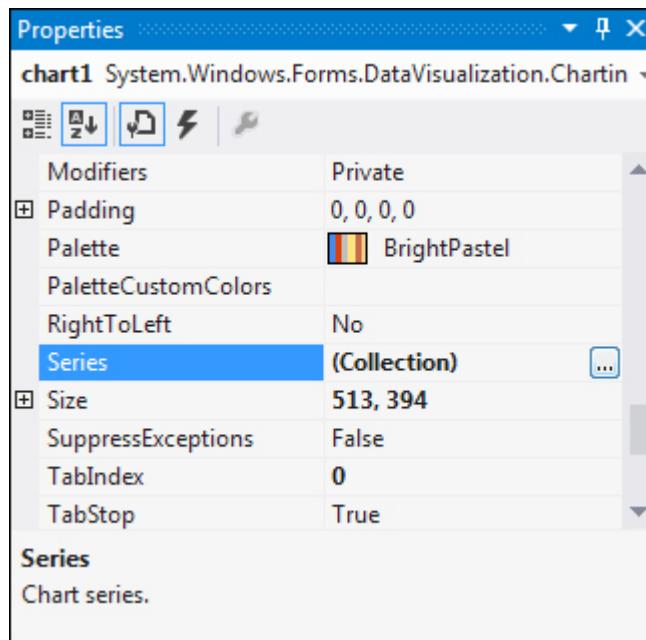
But here's what your chart might look like with a title:



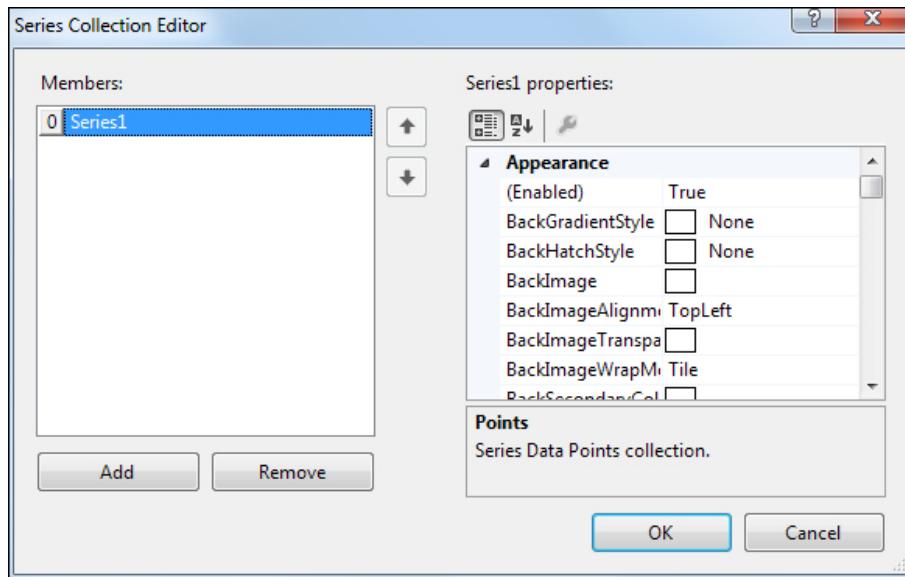
Formatting Chart Series

A series is a collection of data that goes to make up a chart. A series contains things like data points, labels, chart type, and more besides. You can change the appearance of a series quite easily.

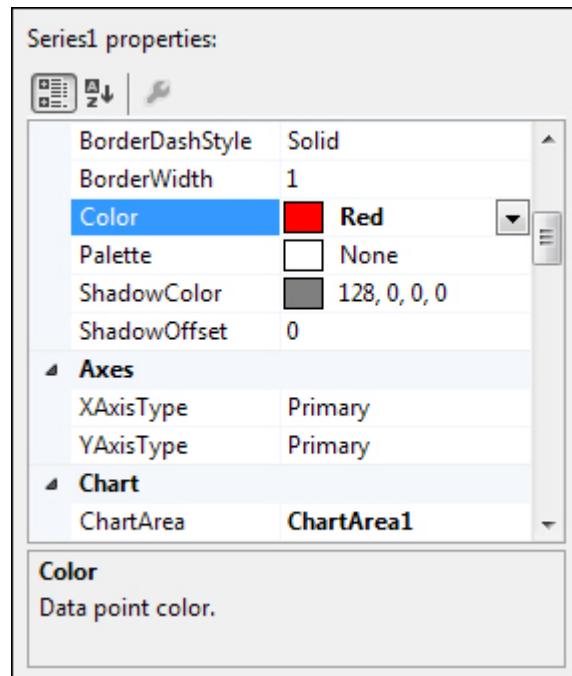
Click on your chart to select it. In the properties area, locate the **Series** item. Click the button to the right of **Collection**:



When you click the button, you'll see the following dialogue box appear:



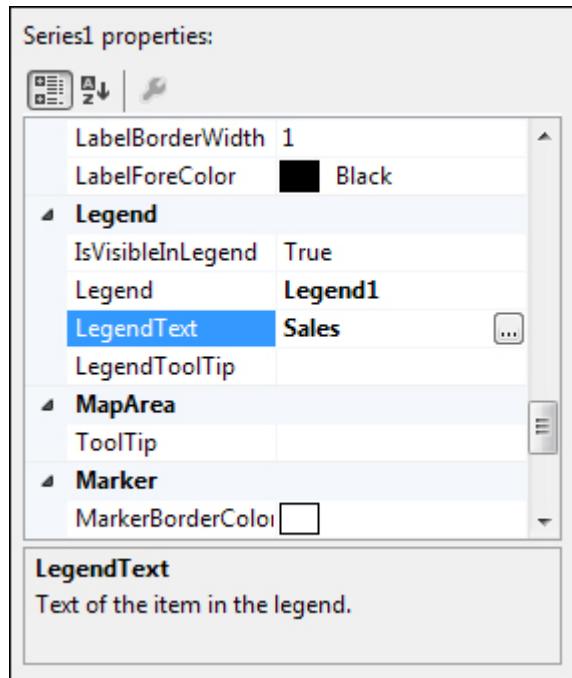
By default, C# adds a Series for you with the name **Series1**. Each Series comes with its own properties. To change the colour of columns in the series, locate the **Appearance** category and select the **Color** property:



Select a colour from the dropdown list. We've gone for Red. As soon as you select a colour, you should see the columns on your chart change. You should also see the Series colour in the Legend area of your chart change as well.

The legend text itself, however, is set to the default of Series1. You'll want to change this to something more descriptive. We want to change our legend text to **Sales**.

To do that, locate the Legend area of the Series1 properties. Under Legend, select **LegendText**. Type Sales into the text area:

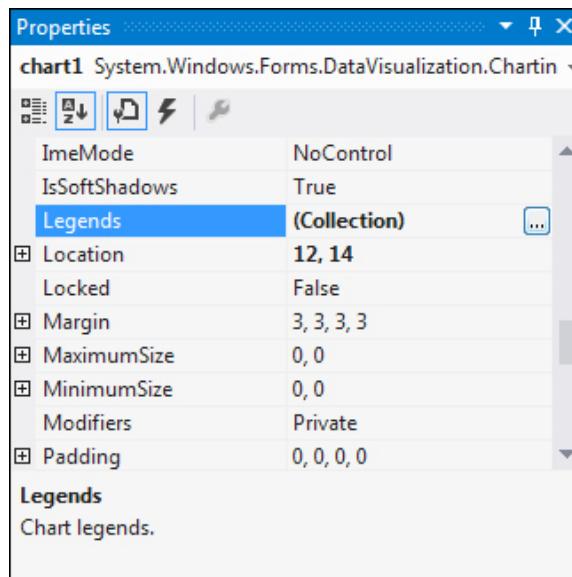


Press enter when you've typed the text and the legend text will change:

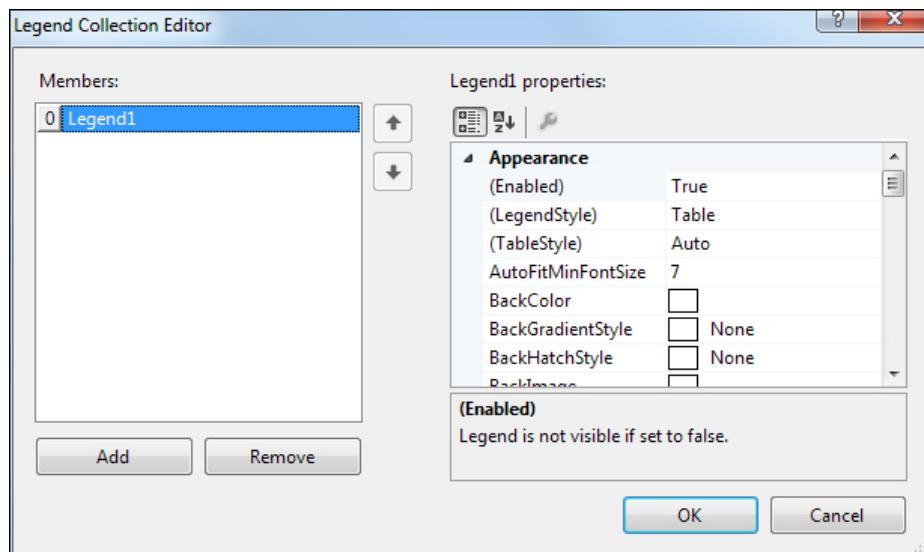


However, the text is rather plain and we'd like to change the font to bold. Rather oddly, you can't do this from the Series dialogue box. You need to access the **Legends** collection for this.

So click OK to get rid of the Series dialogue box. Make sure your chart is selected in the Design environment. In the properties area on the right, locate the **Legends** item:



Click the button to the right of **Collection** to bring up the following dialogue box:



A default Legend of Legend1 has already been added. Each Legend has its own properties. So locate the **Font** property and change it to bold. Locate the **BackColor** property and select a new colour to replace the white one. Locate the **BorderColor** property and change it to anything you like.

If you want, you can add a heading for your Legend. Scroll down the properties and locate the **Title** property. Change it to **Housing**. Change the **TitleFont** property to **Oblique**.

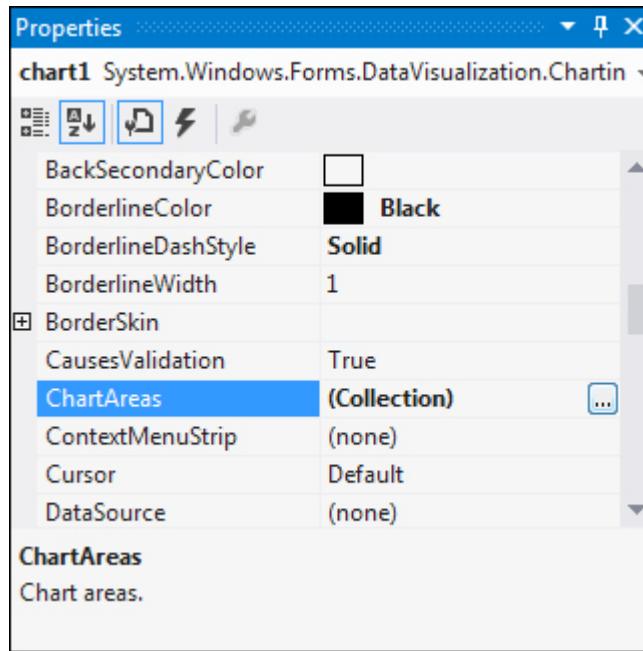
Click OK on the Legend Dialogue Box when you've made the above changes. Your Legend area will then look something like this:



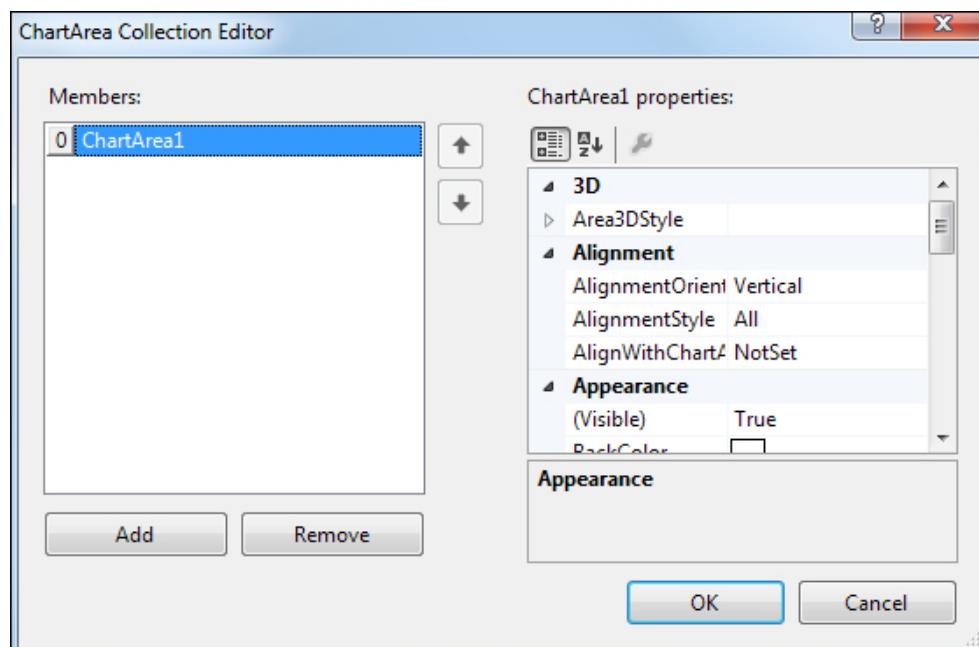
Formatting the ChartArea

The ChartArea can have lots of formatting applied to it as well. Let's see how.

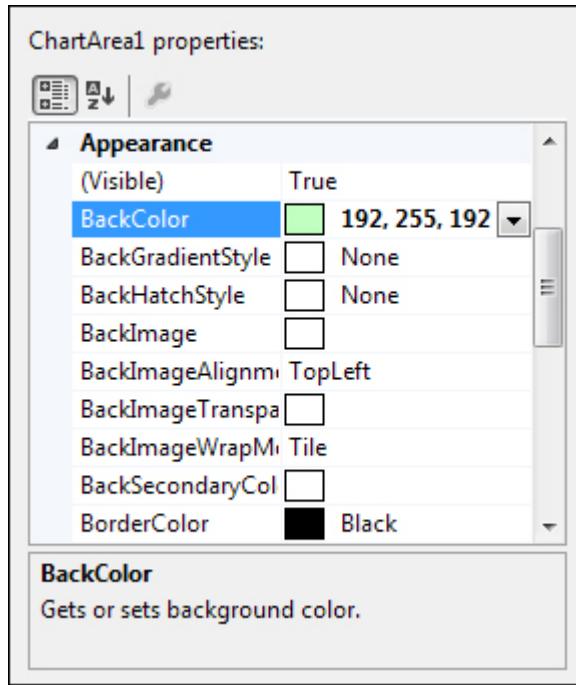
The first thing we'll do is to change the white ChartArea to something else. With your chart selected in Design view, locate **ChartAreas** in the properties box to the right:



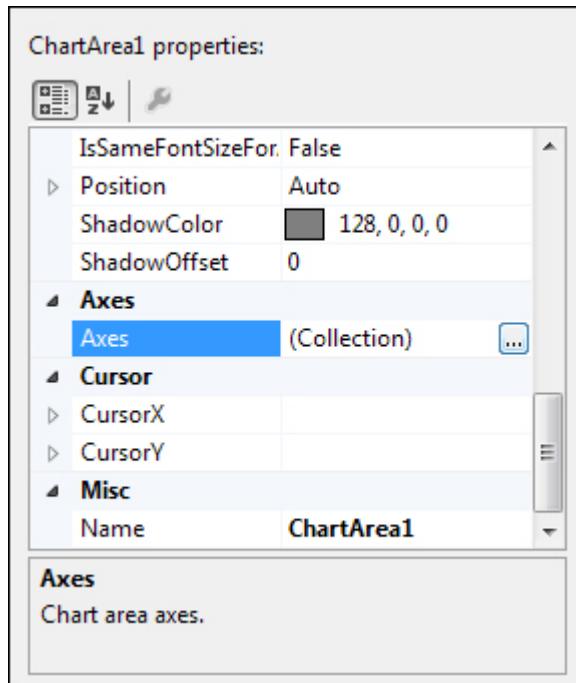
Click the button to the right of **Collection** to see the following dialogue box appear:



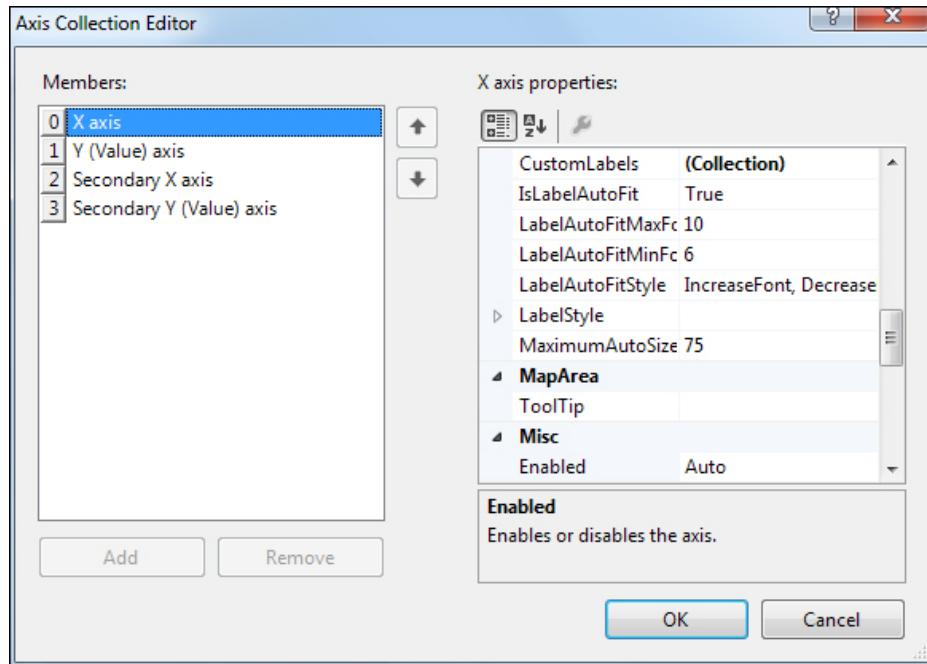
By default, a ChartArea (ChartArea1) has already been added. This has its own properties. To change the colour of the ChartArea, locate the **BackColor** property in the **Appearance** category. Select a new colour for your ChartArea:



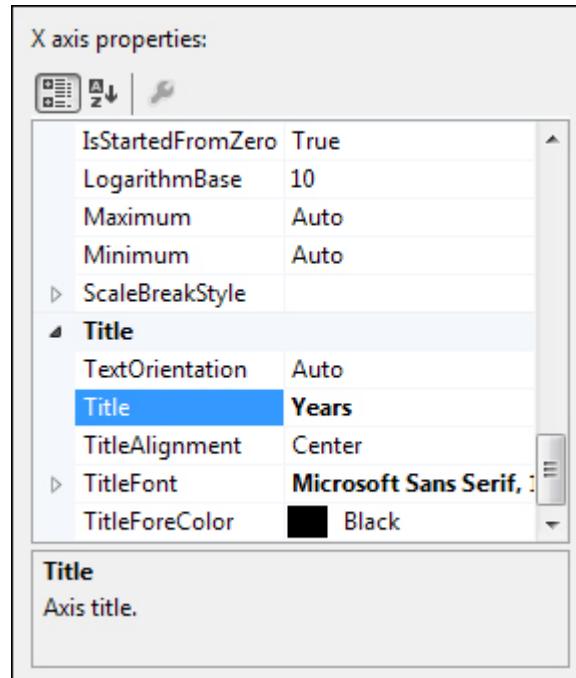
The ChartArea is also where you can set a title for the X and Y Axes. To do this, locate the Axes category:



Click the button to the right of **Collections** to see another dialogue box appear:



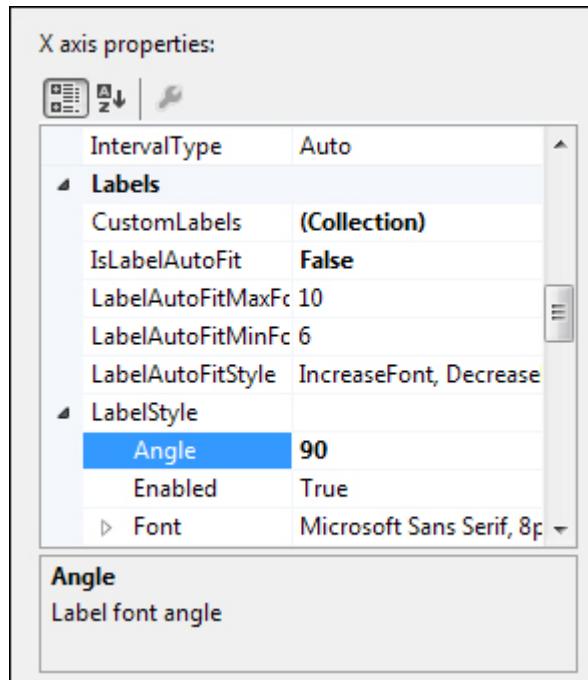
In the members area on the left, you'll see a list of Axes you can change. Make sure the X Axis is selected and have a look at its properties on the right. Locate the **Title** property:



Change the title to **Years**. Change the **TitleFont** while you're there, as well.

One final thing to do. The text for the years goes from left to right. If you want the X Axis titles to have a top to bottom orientation you need to change the **Angle** property.

Scroll up and locate the **Labels** category. In the Labels category, expand the **LabelStyle** item. Locate the **Angle** property and change it to 90:



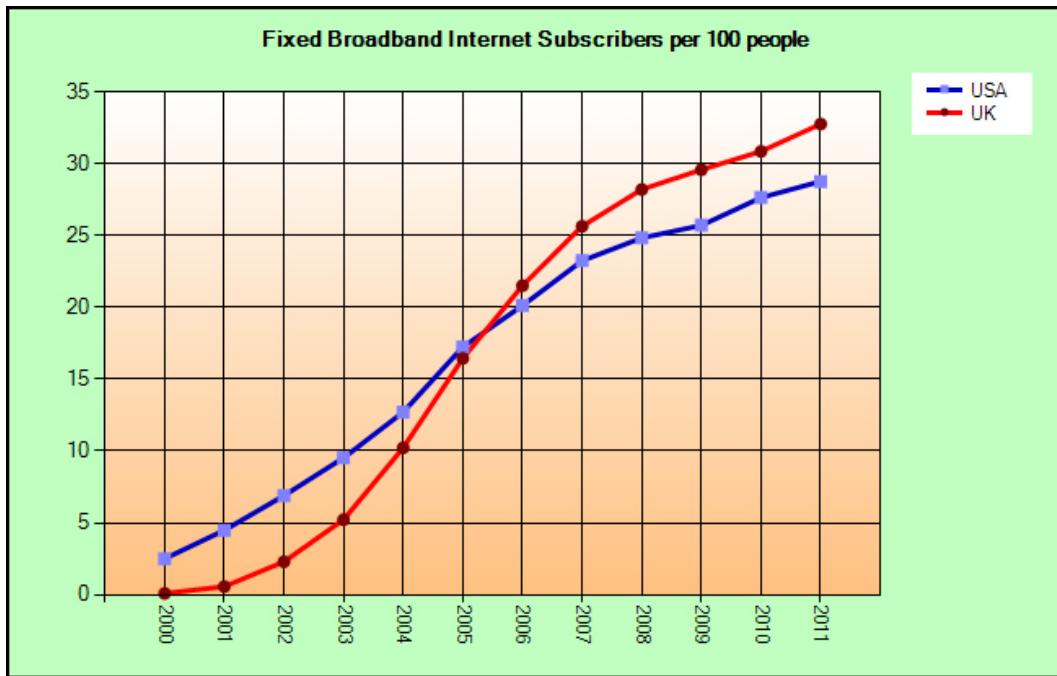
Click OK on all the dialogue boxes to return to Design View. Run your form and your finished chart should look something like this:



OK, we'll now look at other types of chart you can have. We'll start with a line chart.

Line Charts

The next type of chart you'll design looks like this:

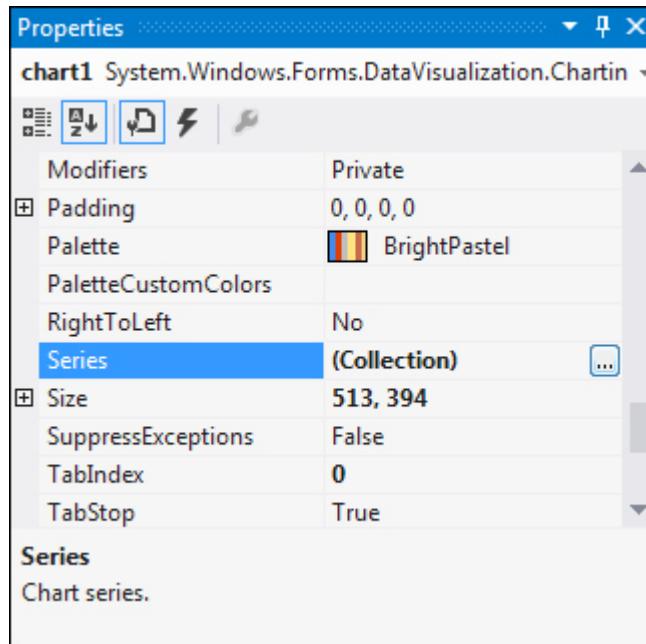


This is a line chart. We have two data series, here, and the chart compares USA and UK fixed line broadband subscribers from the year 2000 up to 2011, per 100 people.

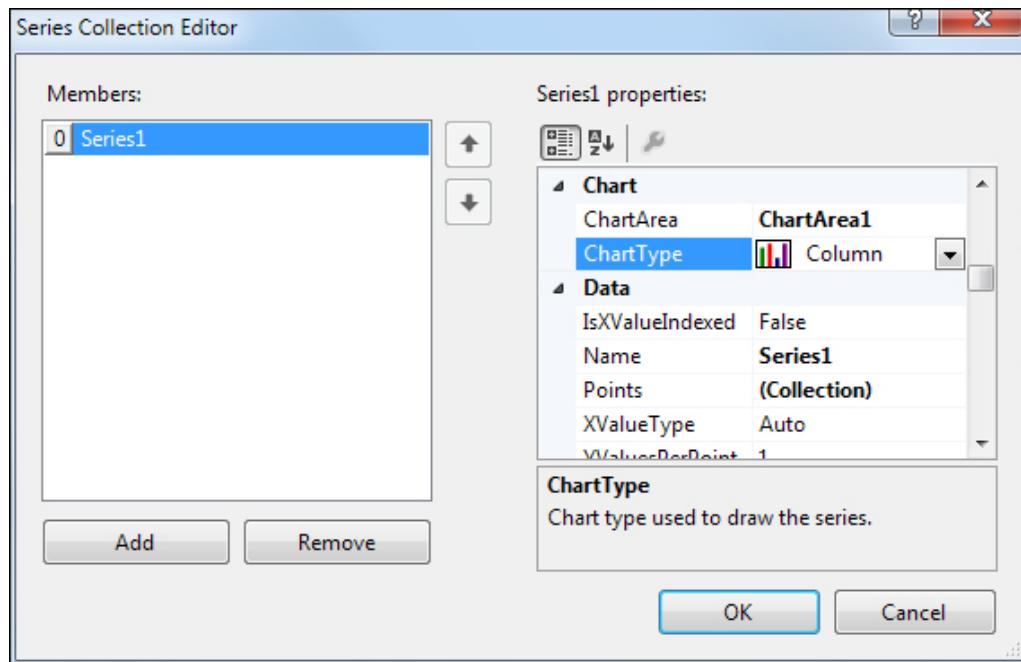
Start a new project for this. Call it **Line_Chart**. Locate the Chart object in the toolbox, under the Data category. Drag and drop a chart onto your new form (or just double click the Chart tool). Make your form and chart a little big bigger.

By default, C# will give you a column chart. We need to change this. Chart Types are part of a Series collection, and each Series can have its own Chart Type.

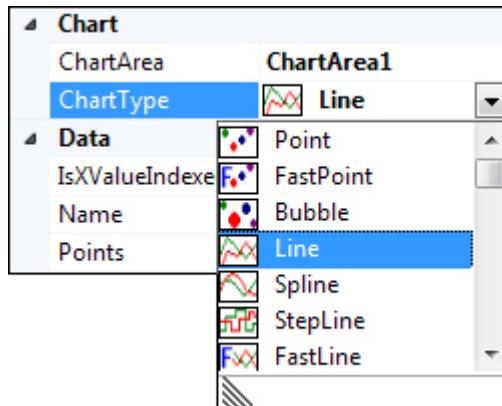
In the Properties area, click on Series to select it. Now click the button to the right of **Collection**:



When you click the Series button, it will bring up the Series Collection Editor dialogue box. A default series called Series1 is already added. Have a look at the properties for Series1 and locate the **ChartType** property:

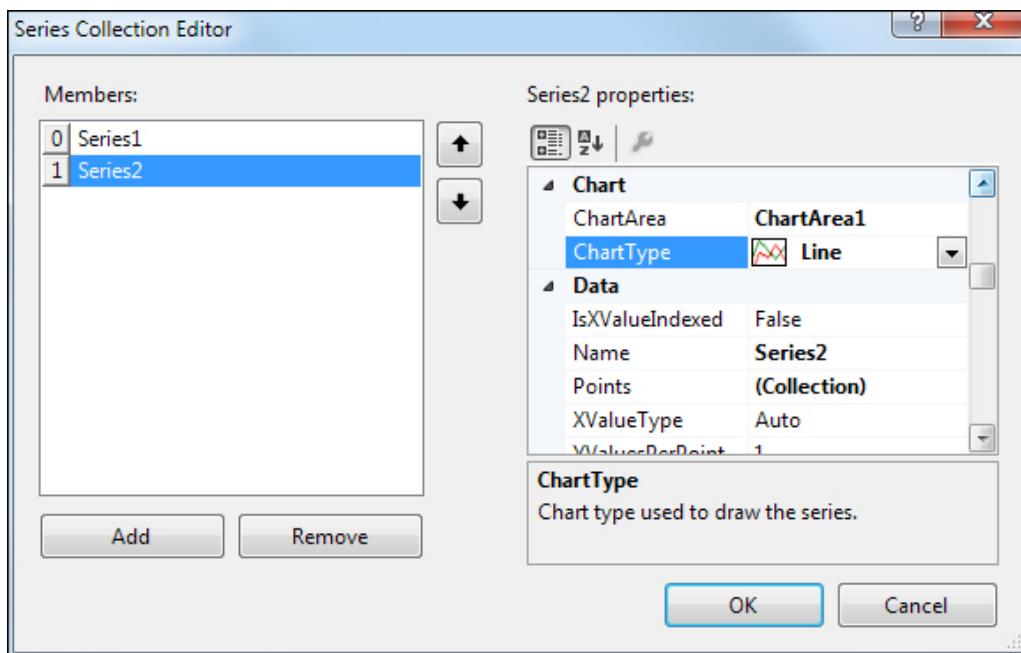


It's set to Column by default. Click the arrow to reveal a dropdown list of charts to choose from. Select **Line**:

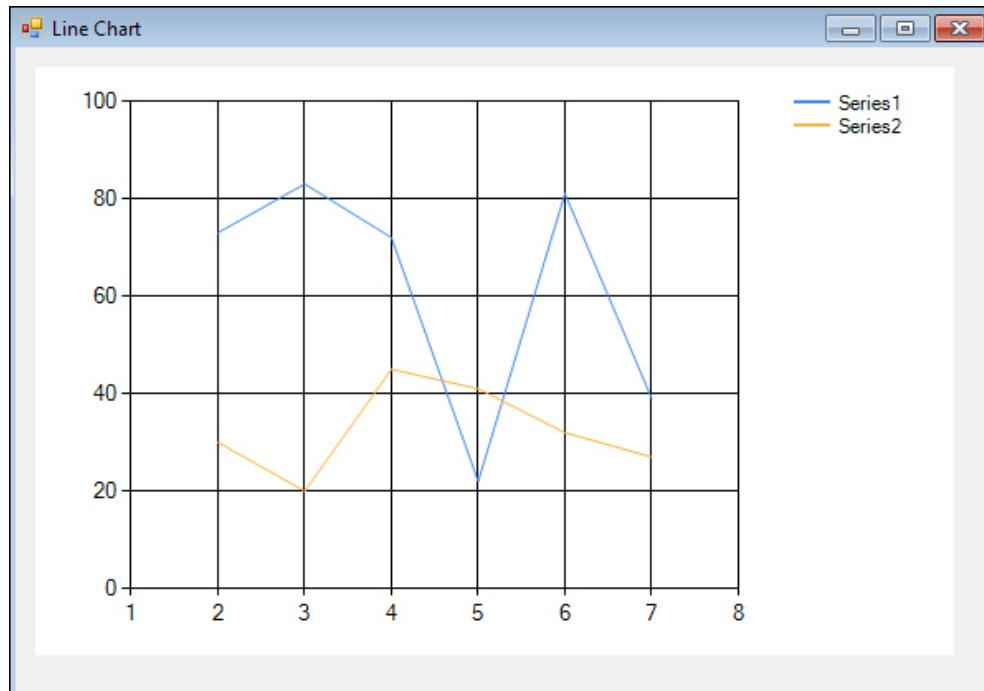


When you select the Line chart option you should see the chart on your form change. It will now have a blue line, representing the data for the chart. This, however, only applies to Series1. If you want a second line on your chart, you need add a new series.

With the Series Collection Editor still displayed, click the **Add** button. This will add a new series to your chart, with the name Series2. Change the **ChartType** for this series to Line, as well:



We'll come back and format the series later. For now, click OK on the Series Collection Editor dialogue box. You should find that your chart now has two line, one blue and one orange:



Although you have two lines on your chart, there is no real data associated with them – just two default lines. To add some data, double click your form (not the chart). This will bring up the code editor and the Form Load event.

Instead of getting data from a text file, like we did for the previous chart, we'll just add two arrays. So add the following two arrays and their values:

```
double[] points_usa = new double[] { 2.50, 4.48, 6.89, 9.52, 12.70,
17.23, 20.11, 23.23, 24.82, 25.69, 27.62, 28.75 };

double[] points_uk = new double[] { 0.09, 0.56, 2.29, 5.22, 10.22, 16.44,
21.50, 25.63, 28.19, 29.56, 30.84, 32.74 };
```

Your code will then look like this:

```
private void Form1_Load(object sender, EventArgs e)
{
    double[] points_usa = new double[] { 2.50, 4.48, 6.89, 9.52, 12.70,
                                         17.23, 20.11, 23.23, 24.82, 25.69, 27.62, 28.75 };

    double[] points_uk = new double[] { 0.09, 0.56, 2.29, 5.22, 10.22,
                                         16.44, 21.50, 25.63, 28.19, 29.56, 30.84, 32.74 };

}
```

For the previous chart, we added data using a loop. There is, however, an easier way – with Data Binding. Add the following two lines to your code:

```
chart1.Series["Series1"].Points.DataBindY(points_usa);
chart1.Series["Series2"].Points.DataBindY(points_uk);
```

The first line specifies a chart called **chart1**. After a dot we have an item in a series collection with the name of a particular series in square brackets. We then access the Points collection, like we did for a column chart. This time, though, notice the **DataBindY** part:

```
DataBindY( points_usa );
```

DataBindY means uses data binding for the Y Axis values. In between round brackets, we have our arrays. You can use data binding to bind things like lists, arrays, data tables, data sets, and a whole lot more besides. This saves you from having to go round in a loop adding points to a series.

Your code should now look like this, though:

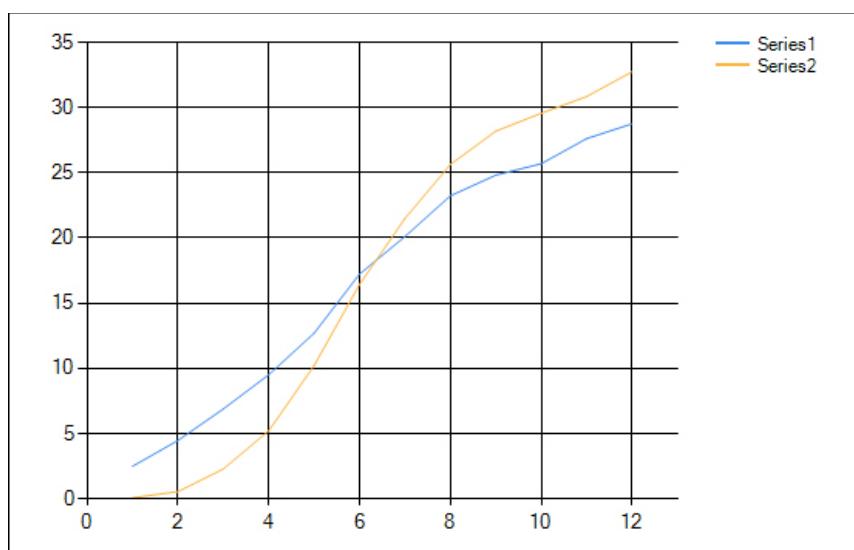
```
private void Form1_Load(object sender, EventArgs e)
{
    double[] points_usa = new double[] { 2.50, 4.48, 6.89, 9.52, 12.70,
                                         17.23, 20.11, 23.23, 24.82, 25.69, 27.62, 28.75 };

    double[] points_uk = new double[] { 0.09, 0.56, 2.29, 5.22, 10.22,
                                         16.44, 21.50, 25.63, 28.19, 29.56, 30.84, 32.74 };

    chart1.Series["Series1"].Points.DataBindY(points_usa);
    chart1.Series["Series2"].Points.DataBindY(points_uk);

}
```

You can run your programme now and test it out. You should find that your chart looks like this:



The values on the Y Axis are OK: they go from 0 to 35, representing the number of people per 100 who have subscribed to fixed line broadband. But the values on the X Axis are not the values we want. Like our previous chart, we want dates on the X Axis.

You can add the dates in exactly the same way you did for your column chart. Here's the code:

```
string[] dates = new string[12] {"2000","2001","2002","2003",
"2004", "2005", "2006", "2007", "2008", "2009", "2010", "2011" };

CustomLabel customLabel;
Axis axisX = chart1.ChartAreas[0].AxisX;

int endLoop = dates.Length;
double axelLabelPos = 0.5;

for (int i = 0; i < endLoop; i++)
{
    customLabel = axisX.CustomLabels.Add(axelLabelPos,
    axelLabelPos + 1, dates[i]);

    axelLabelPos = axelLabelPos + 1.0;
}
```

Before adding it, however, you'll need a new using statement at the top:

```
using System.Windows.Forms.DataVisualization.Charting;
```

Your coding window will then look like this on the next page:

```

using System.Windows.Forms.DataVisualization.Charting;

namespace LineChart
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            double[] points_usa = new double[] { 2.50, 4.48, 6.89, 9.52, 12.70,
                17.23, 20.11, 23.23, 24.82, 25.69, 27.62, 28.75 };

            double[] points_uk = new double[] { 0.09, 0.56, 2.29, 5.22, 10.22,
                16.44, 21.50, 25.63, 28.19, 29.56, 30.84, 32.74 };

            chart1.Series["Series1"].Points.DataBindY(points_usa);
            chart1.Series["Series2"].Points.DataBindY(points_uk);

            string[] dates = new string[12] { "2000", "2001", "2002", "2003", "2004", "2005",
                "2006", "2007", "2008", "2009", "2010", "2011" };

           CustomLabel customLabel;
            Axis axisX = chart1.ChartAreas[0].AxisX;

            int endLoop = dates.Length;
            double axelLabelPos = 0.5;

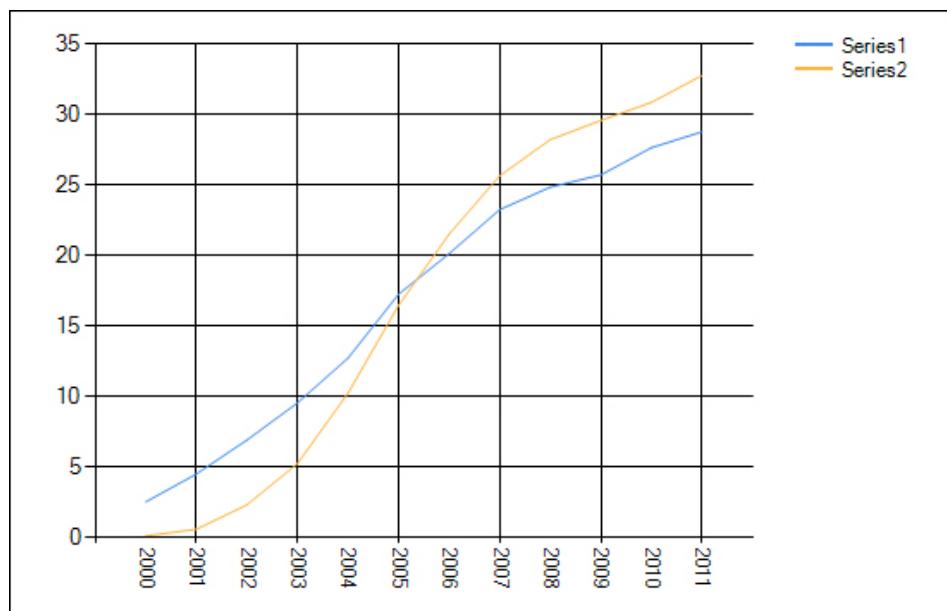
            for (int i = 0; i < endLoop; i++)
            {
                customLabel = axisX.CustomLabels.Add(axelLabelPos, axelLabelPos + 1, dates[i]);

                axelLabelPos = axelLabelPos + 1.0;
            }
        }
    }
}

```

If you were to run your programme now, you'd find that the dates on the X Axis are not rotated. Rotate them 90 degrees, like you did with the column chart. (From the properties area its **ChartAreas > Axes > X Axis > Label Style > Angle: 90.**)

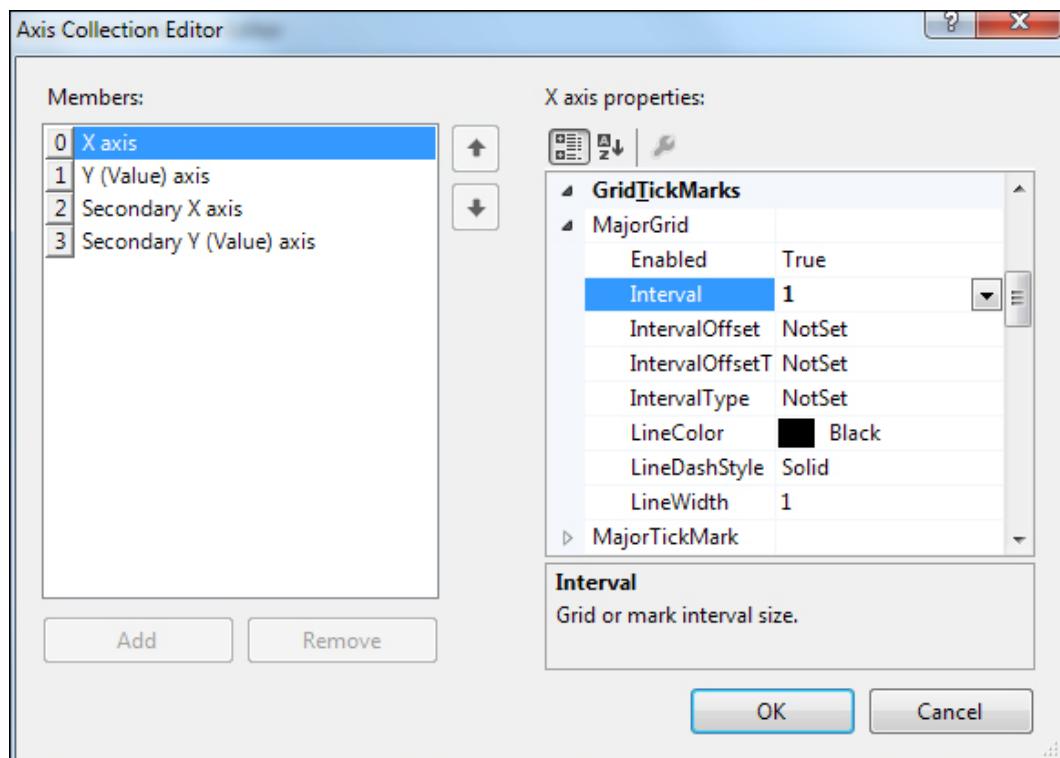
Once you've rotated the labels on the X Axis your chart will look like this:



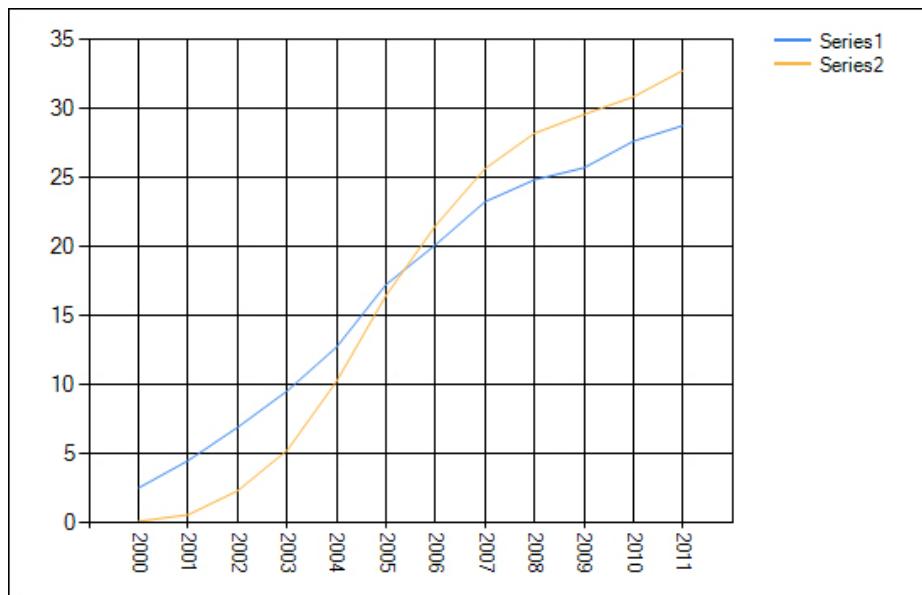
If you notice the vertical lines, however, you'll see that there are none for the years 2000, 2002, 2004, 2006, 2008, and 2010. We can fix that now.

Chartareas > Axes > X Axis > Grid tick marks > Major grid > Interval: 1

With your chart selected in Design View, click on the **ChartAreas** property. Click the button to the right of **Collections** to bring up the **ChartAreas Collection Editor**. Locate the **Axes** property and click the button to the right of Collections. This will bring up the **Axes Collection Editor**. Make sure the X Axis is selected in the members area on the left. In the properties area on the right, locate the **Grid Tick Marks** section. For the **Major Grid** property set the Interval to 1:

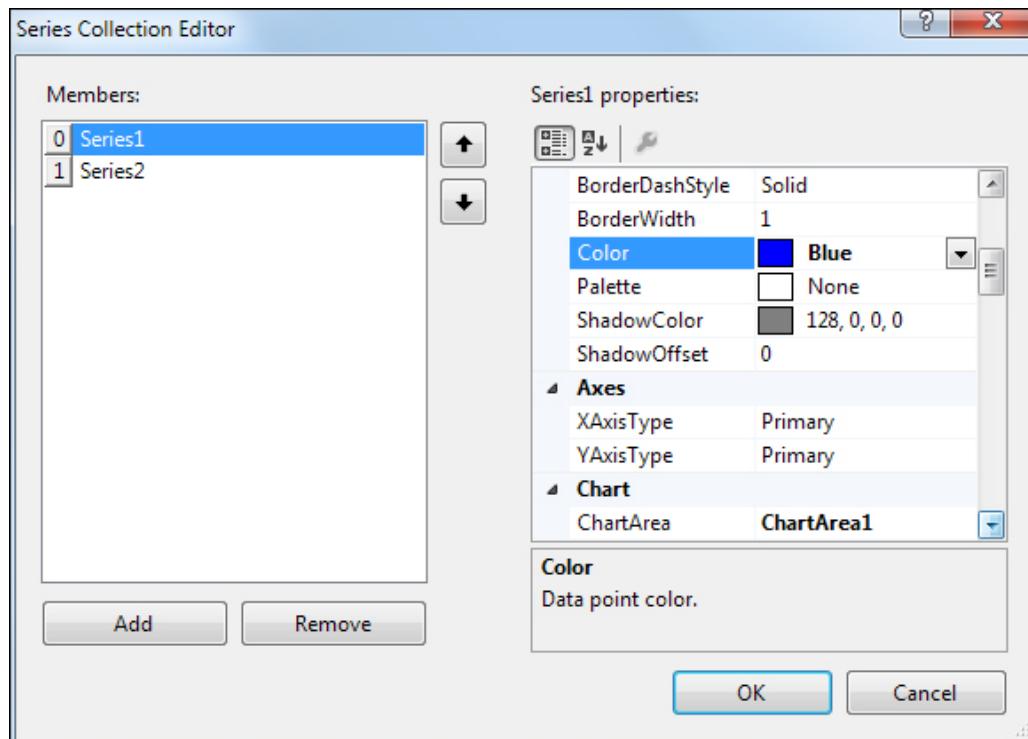


Click OK on all the dialogue boxes to get rid of them. Now run your form again. You should see that the missing vertical lines for the dates are now there:



The next thing we can do is to change the colours of the lines in the series. We can also add some markers for each point on the line.

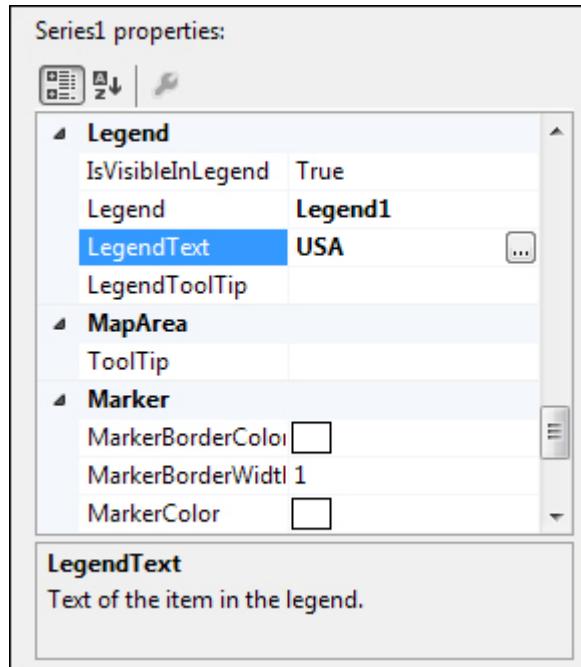
Bring up the Series Collection Editor again. With Series1 selected in the **Members** area on the left, have a look at the properties on the right. Locate the **Color** property in the **Appearance** category:



Select a colour for your first series.

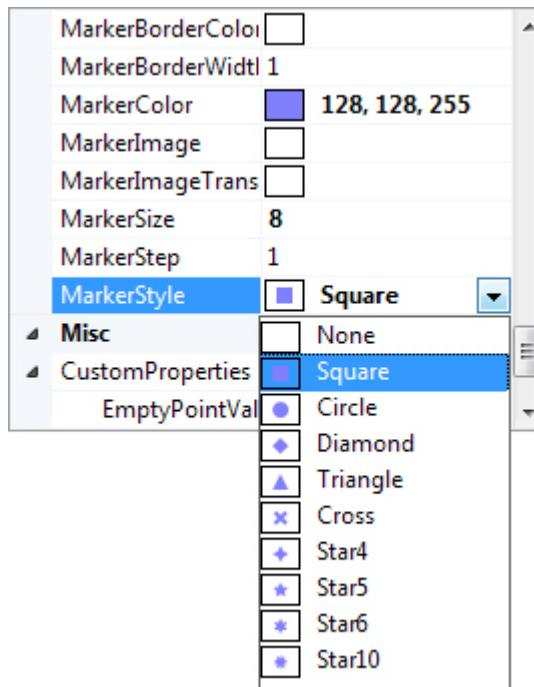
To make the series line a bit thicker, change the **BorderWidth** property (just above Color) to 3.

To change the Legend from the default Series1 to something more appropriate, scroll down and locate the **Legend** category. Type USA for the **LegendText** property:



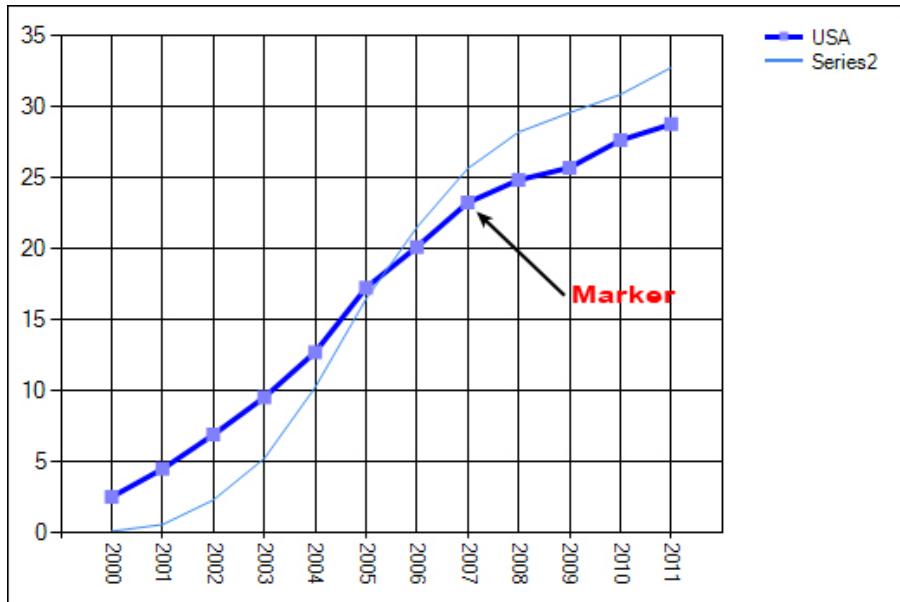
Adding Markers

A Marker is a data point indicator that is used to join your lines. If that's not too clear, scroll down and locate the **Marker** category. Click the arrow to reveal a drop down list of Marker styles you can use:



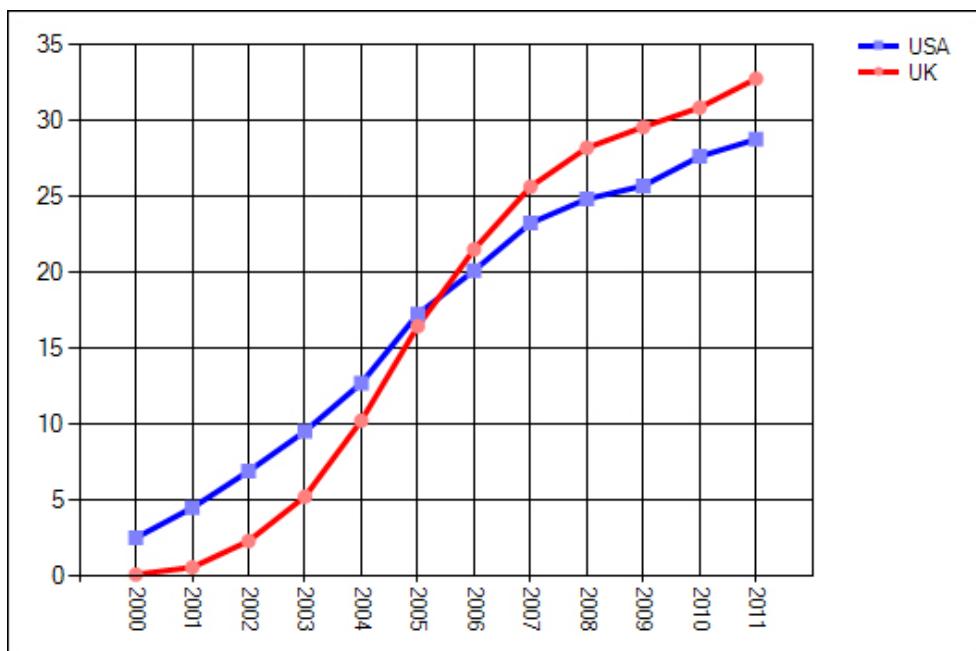
We've selected a Square marker, here. We've also changed the **MarkerSize** property to 8 and chosen a **MarkerColor**. Make the changes to your own Series1, as well.

When you make the changes, your form will look like this:



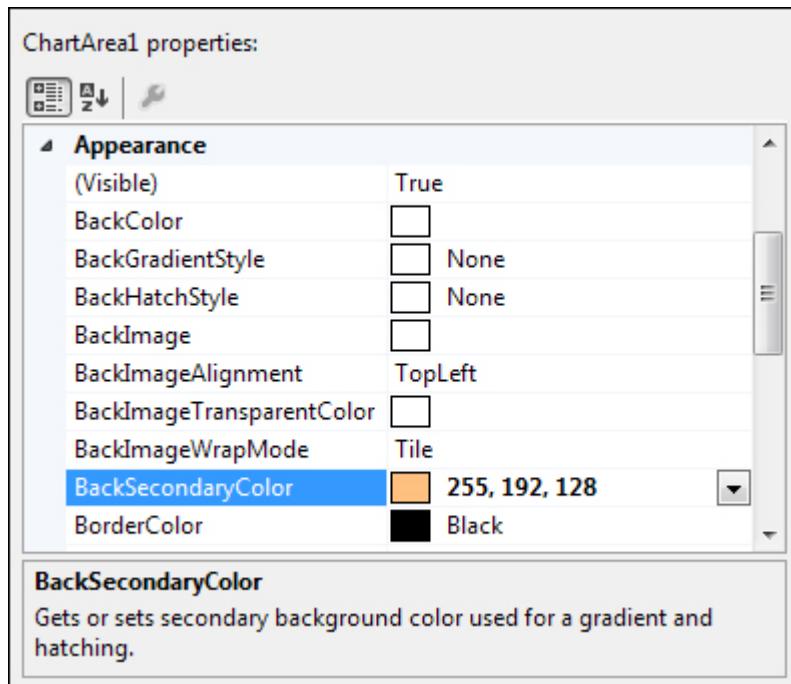
The markers correspond to the values from the array we set up in the Form Load event.

When you've made the changes to Series1, click on Series2 in the members area on the left. Now make similar changes to the properties for this series. Use different colours, though. And set the **MarkerStyle** to Circle. For the Legend Text, type UK. Your form will then look something like ours below:

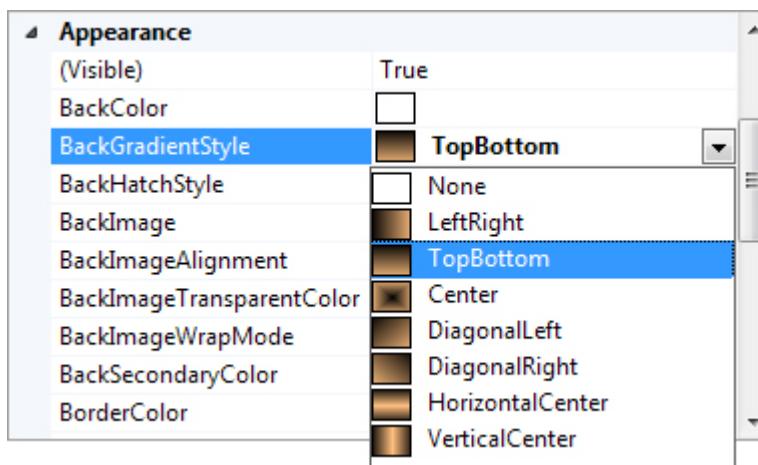


You can have an image or gradient behind your charts. We'll add a gradient.

In Design View, click on your chart to select it. In the properties area, locate the **ChartAreas** item. Then click the button to the right of **Collection**. This will bring up the ChartAreas Collection Editor. In the properties area to the right of the ChartAreas Collection Editor locate the **BackSecondaryColor** item:

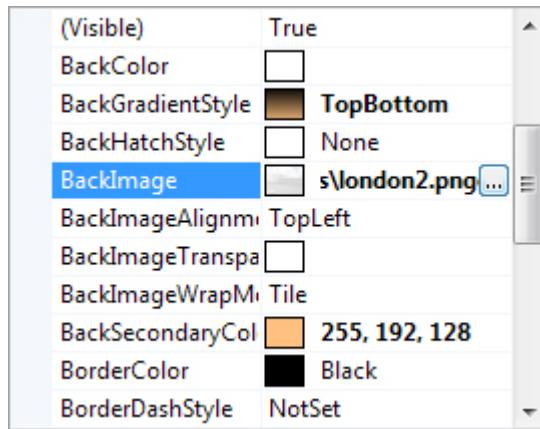


Select a colour. We've gone for orange. To get a gradient, expand the **BackGradientStyle** item:



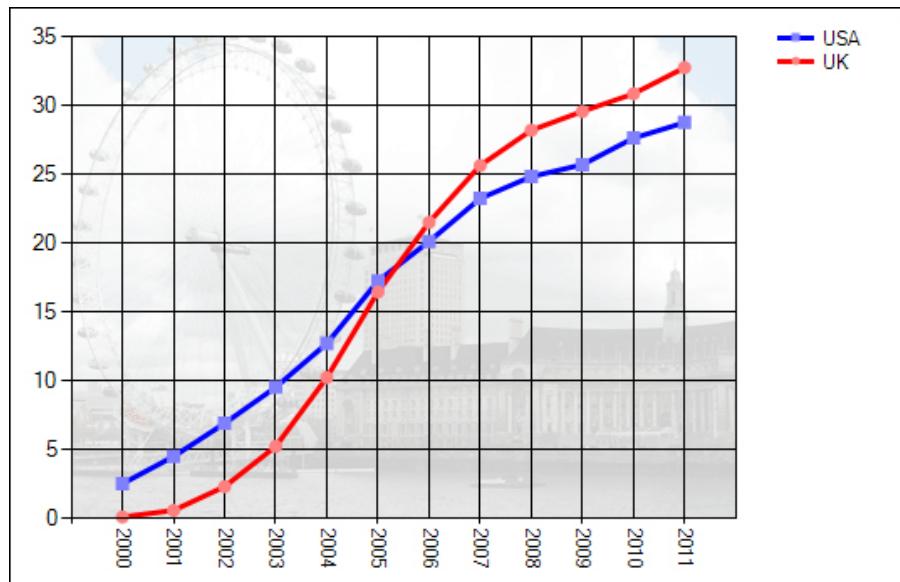
Select a gradient from the list. We've chosen **TopBottom**.

If you prefer, you can select an image instead of gradient. This is done via the **BackImage** property, two down from the **BackGradientStyle** item above:

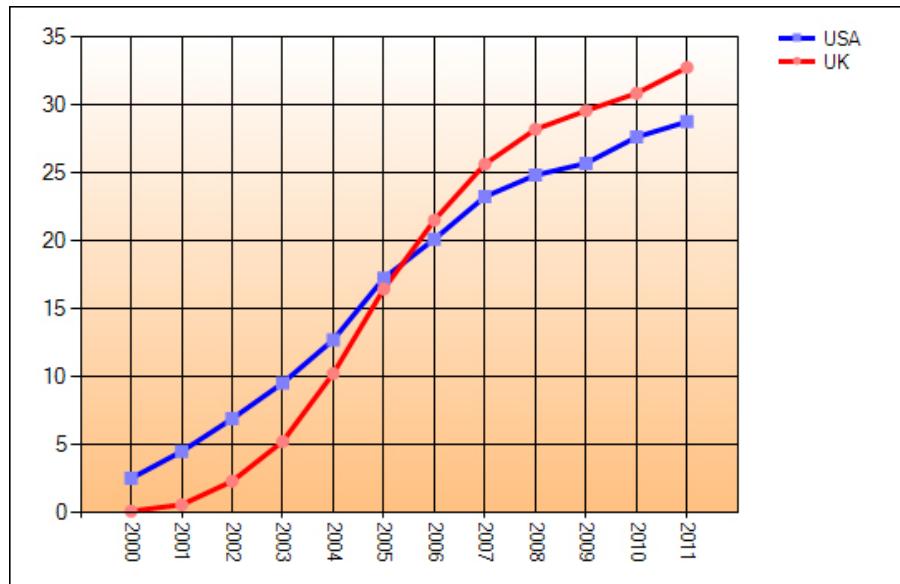


Click the button to the right of the **BackImage** item and then search for a picture on your computer.

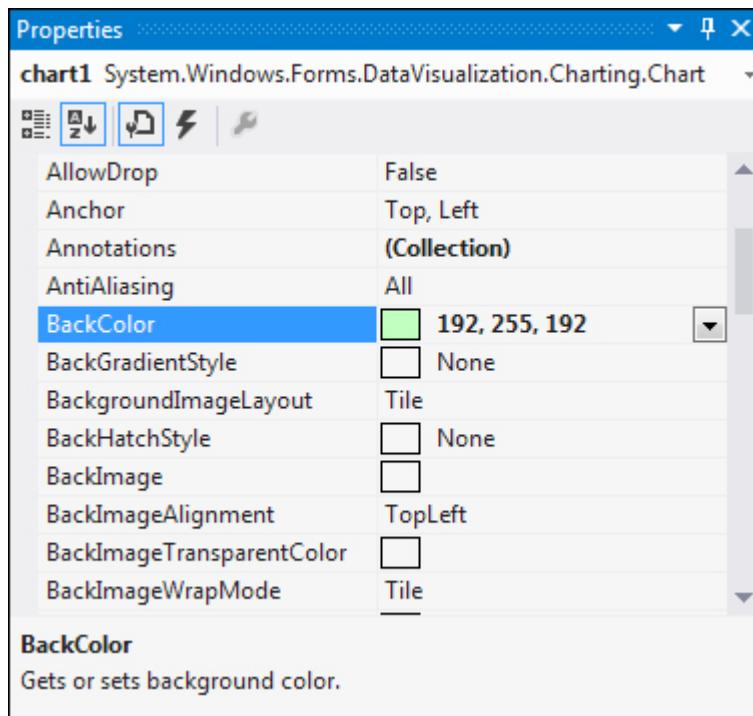
Whether you go for an image or a gradient, though, click OK on the ChartAreas Collection Editor. Your chart might look like this if you chose an image:



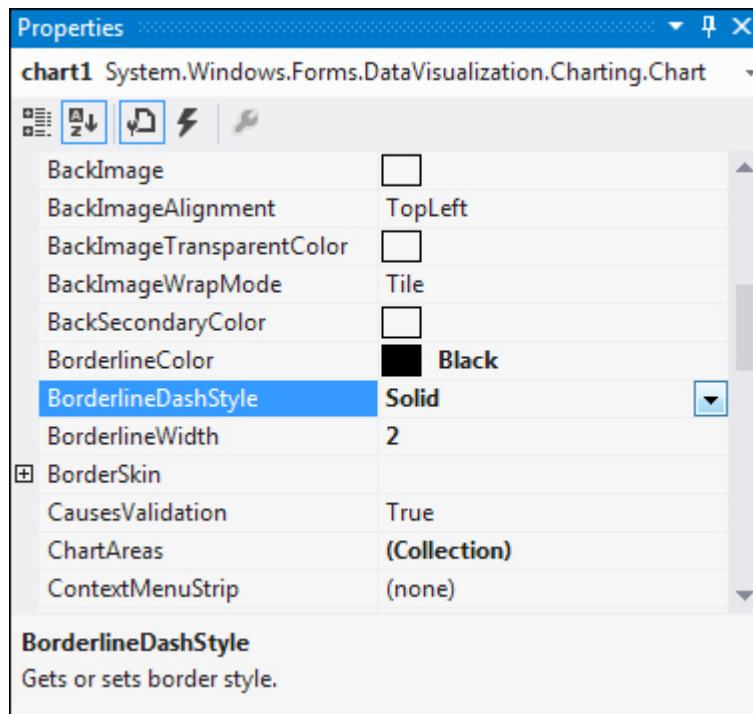
Or this if you went for a gradient:



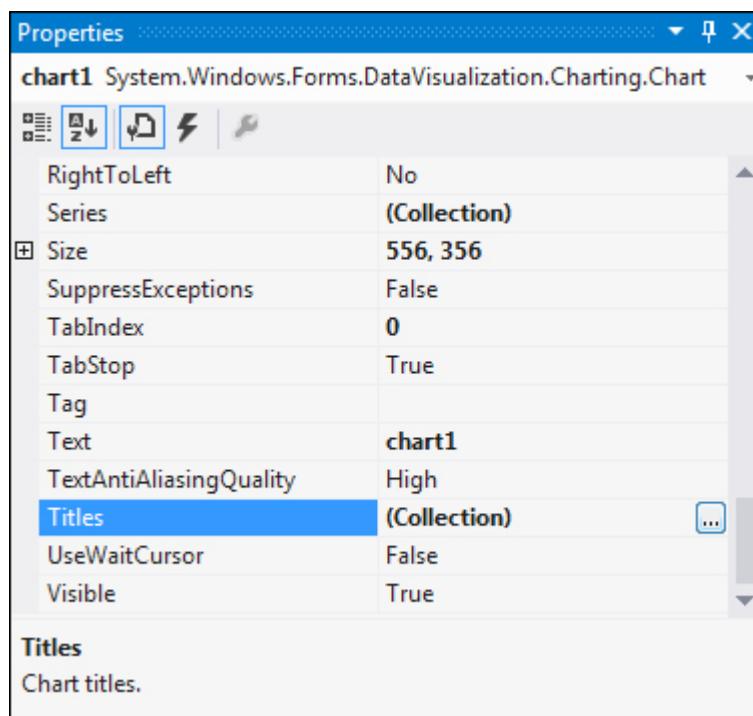
Not much left to do on this chart, now. To change the colour of the whole chart (the white parts), make sure your chart is selected in Design View. In the properties area, locate the **BackColor** property. Change it to any colour you like:



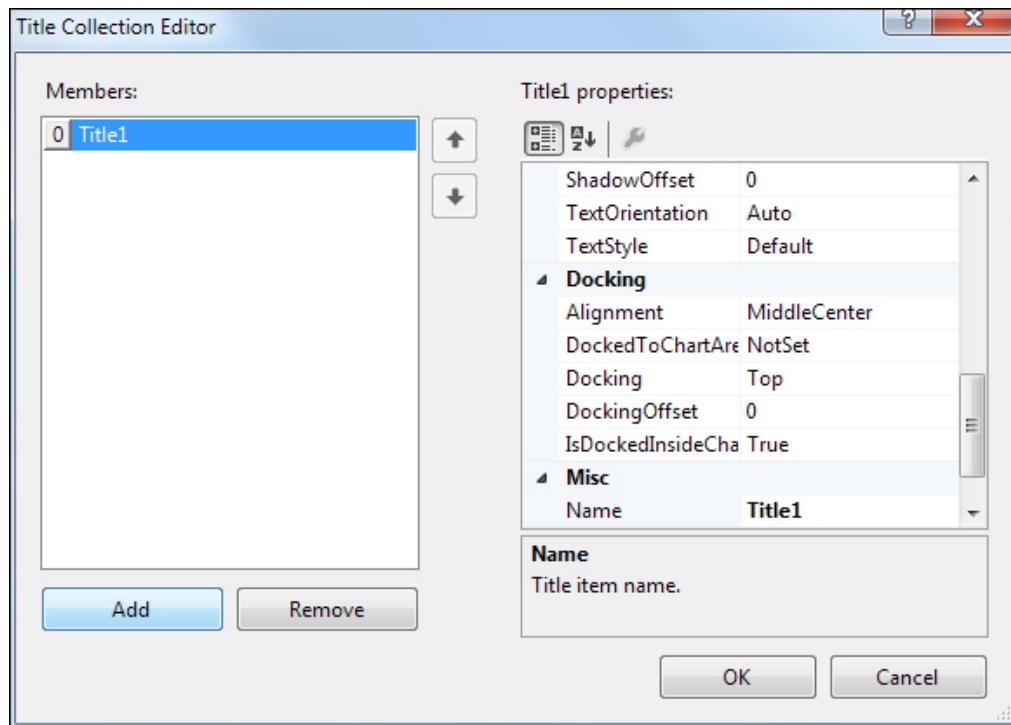
If you want to set a solid border around the whole of your chart, change the **BorderlineDashStyle** property to Solid, the **BorderlineWidth** to 2 (or any value you like), and set a **BorderlineColor**:



To add a title, scroll down in the properties area and locate the **Titles** collection:

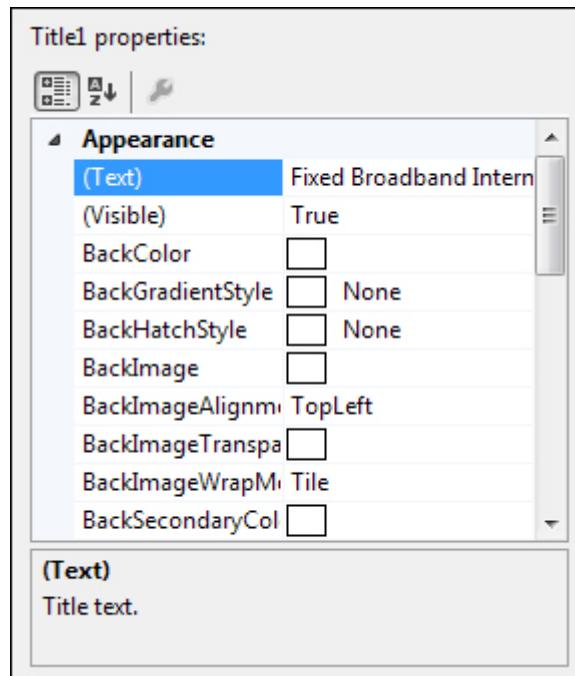


Click the button to bring up the Titles Collection Editor. Click the Add button to add a new title:

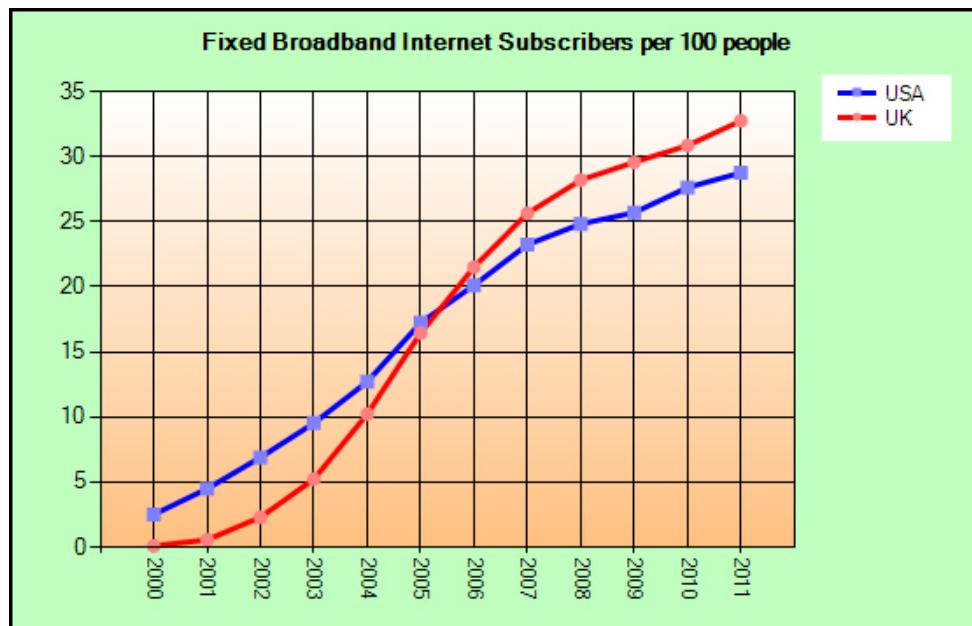


In the properties area for your new title, locate the text property at the top. Type the following title:

Fixed Broadband Internet Subscribers per 100 people



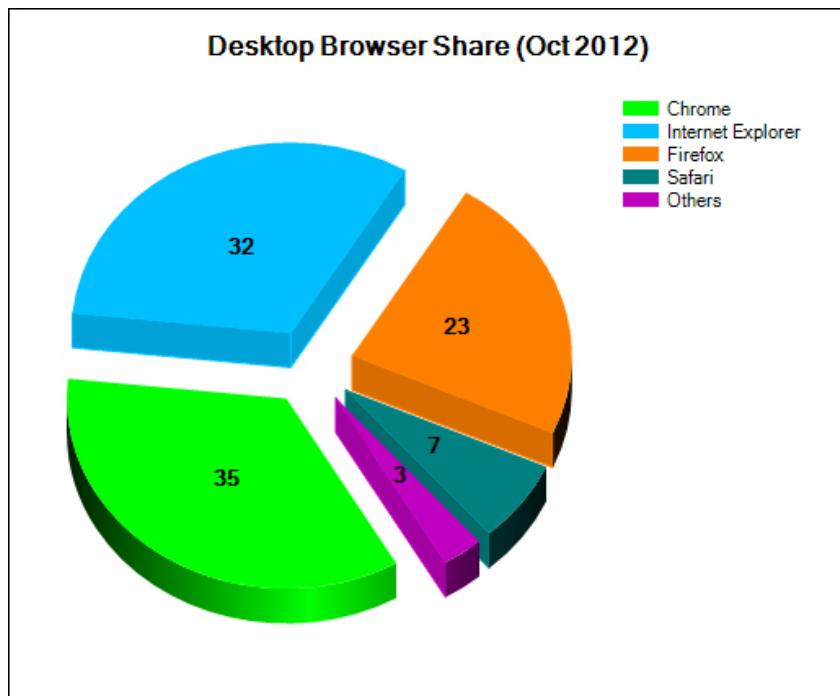
Now scroll down and locate the Font property. Change it any font style you like. Click OK on the Titles Collection Editor to return to Design View. Run your programme and your final chart will look like this:



OK, that's it for line charts. We'll take a look at one more chart style before moving on: Pie Charts.

Pie Charts

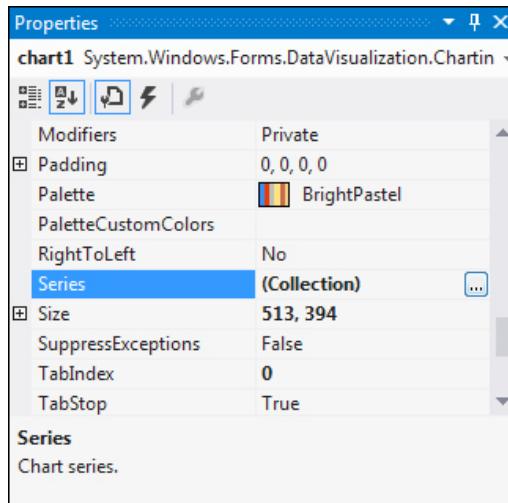
The final chart we'll take a look at is a pie chart. It will look like this when we're finished:



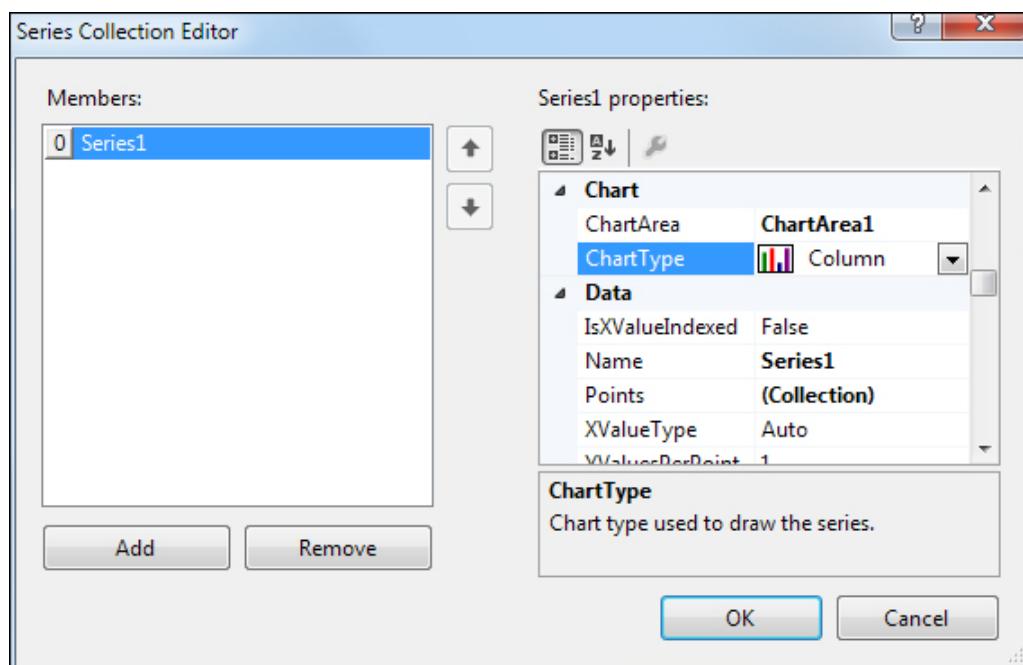
Start a new project for this. Call it **PieChart**. Locate the Chart object in the toolbox, under the Data category. Drag and drop a chart onto your new form (or just double click the Chart tool). Make your form and chart a little bigger.

By default, C# will give you a column chart. We need to change this. Chart Types are part of a Series collection, and each Series can have its own Chart Type.

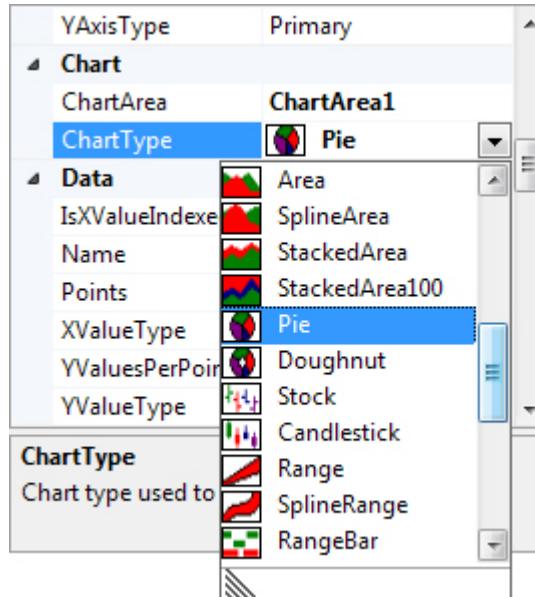
In the Properties area, click on Series to select it. Now click the button to the right of **Collection**:



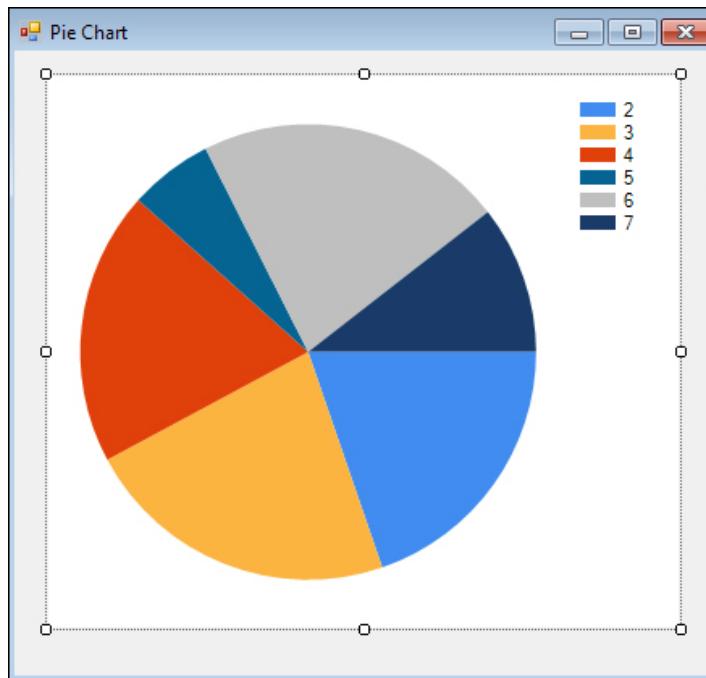
When you click the Series button, it will bring up the **Series Collection Editor** dialogue box. A default series called Series1 is already added. Have a look at the properties for Series1 and locate the **ChartType** property:



It's set to Column by default. Click the arrow to reveal a dropdown list of charts to choose from. Select the **Pie** option:



When you select the Pie chart option you'll see your form change to this:



This is a default pie chart, with no real data associated with it. Let's add some data.

Adding Data for a Pie Chart

Pie chart data consists of values for each slice of the pie, and a series to the right explaining what each slice represents. Our pie chart will show which desktop

browsers were the most popular for a particular month of the year. The values for each slice are these:

35, 32, 23, 7, 3

The browsers are these:

"Chrome", "Internet Explorer", "Firefox", "Safari", "Others"

We'll set these up as arrays.

Double click your form (not the chart) to open up the Form Load event. Add the following two arrays:

```
double[] pie_points = new double[] { 35, 32, 23, 7, 3 };

string[] series_text = new string[] { "Chrome", "Internet Explorer",
    "Firefox", "Safari", "Others" };
```

The first array is of type double, while the second array is of type string.

We can use data binding again to attach the values and the strings as data points. So add the following line:

```
chart1.Series["Series1"].Points.DataBindXY(series_text, pie_points);
```

This is similar to the code you used for the line chart. The only difference is this:

```
DataBindXY( series_text, pie_points );
```

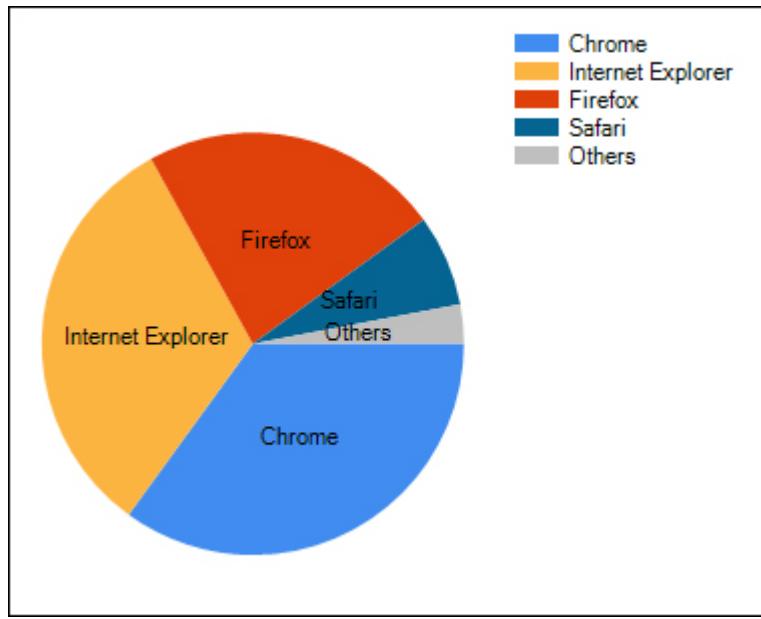
This time we are using **DataBindXY**. For the line chart we only needed to data bind to the Y column. For the pie chart we need to bind the text and the series data. In between the round brackets of **DataBindXY** we first need the series text (the X values), After a comma you then type your data (the Y values).

Your code should now look like this:

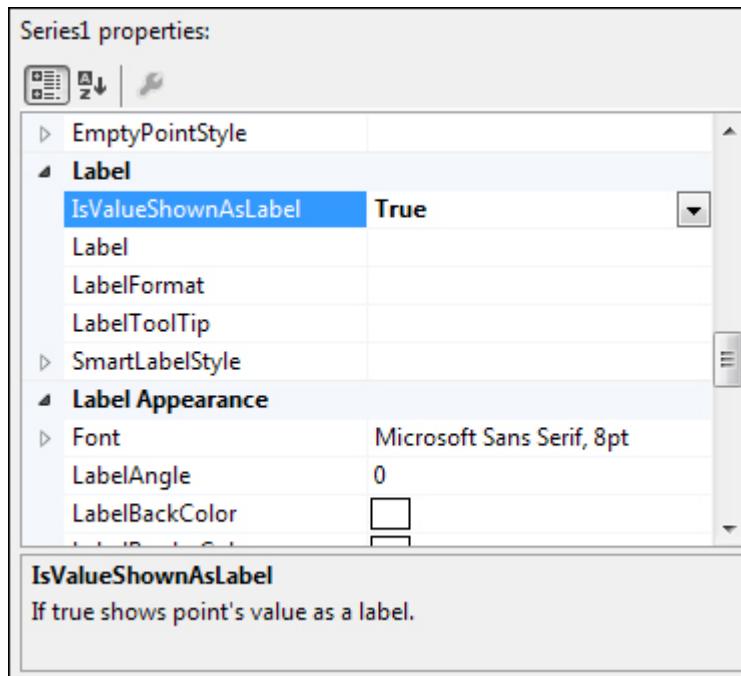
```
private void Form1_Load(object sender, EventArgs e)
{
    double[] pie_points = new double[] { 35, 32, 23, 7, 3 };
    string[] series_text = new string[] { "Chrome", "Internet Explorer",
        "Firefox", "Safari", "Others" };

    chart1.Series["Series1"].Points.DataBindXY(series_text, pie_points);
}
```

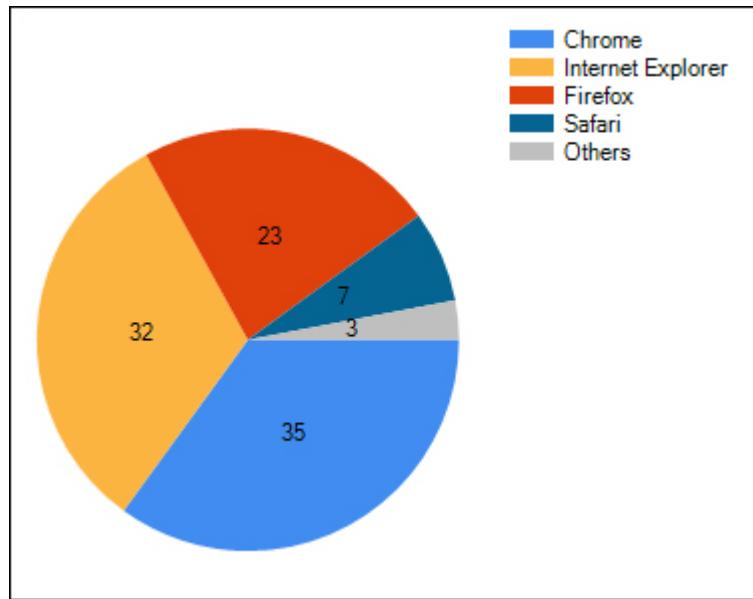
When you run your programme, however, you should see this:



Only the text has displayed on the slices, and no values. To solve this, stop your programme and return to Design View. Click your chart to select it. From the properties area on the right, bring up the Series Collection Editor (you should know how to do this by now). From the Series Collection Editor locate the **IsValueShownAsLabel** property, under the Label category. Set it to **True**:



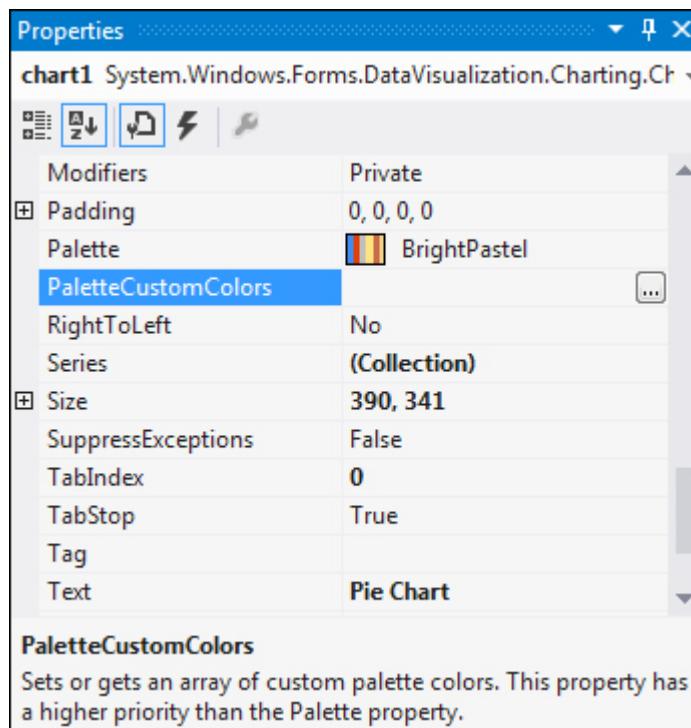
Run your programme again and you should find that the value are displayed on the pie slices:



Series Custom Colours

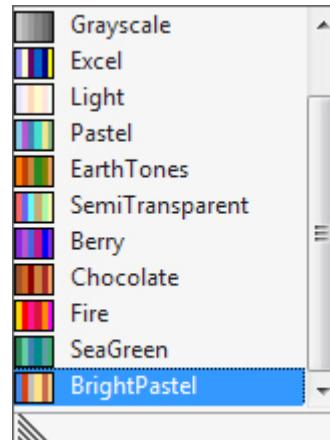
The colours used for the series are not quite what we want. We want Internet Explorer to be blue and chrome to be green. We'd like to set Firefox to orange and Safari to green. For all other browser, we'd like this as purple. To do this, C# allows you to set up custom colours for your chart series.

In Design View, click on your chart to select it. In the properties area on the right locate the **PaletteCustomColors** item:



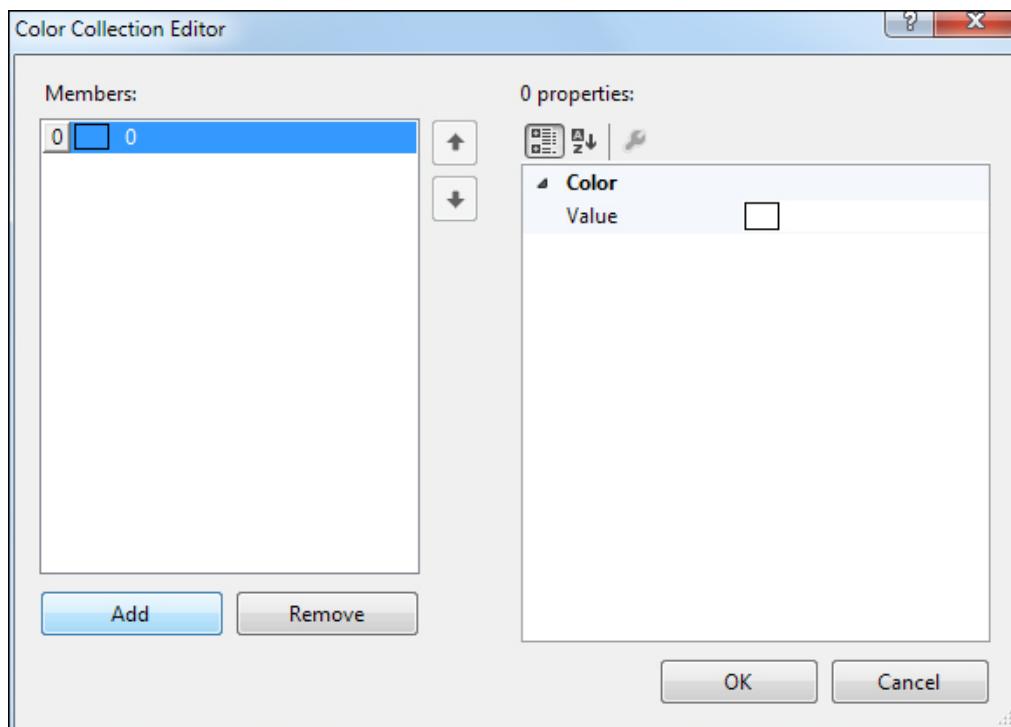
The text at the bottom says, "Sets or gets an array of custom palette colors. This property has a higher priority than the Palette property".

The Palette property it's talking about is the one just above the **PaletteCustomColors** item – **Palette**. Click the down arrow on the Palette property to see a list of colours already set up:

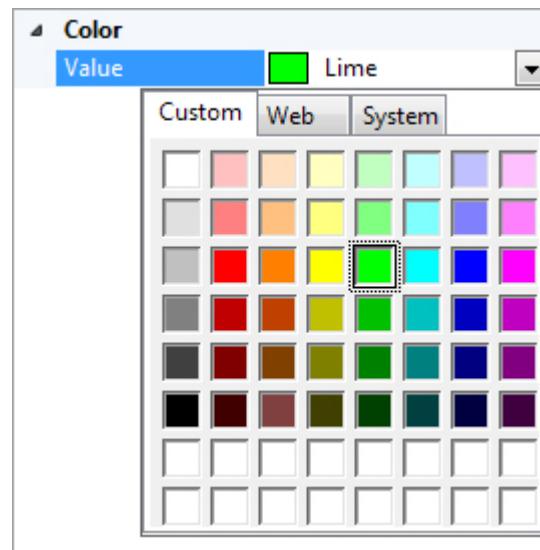


If you liked a set of these colours, you could simply select one from the list.

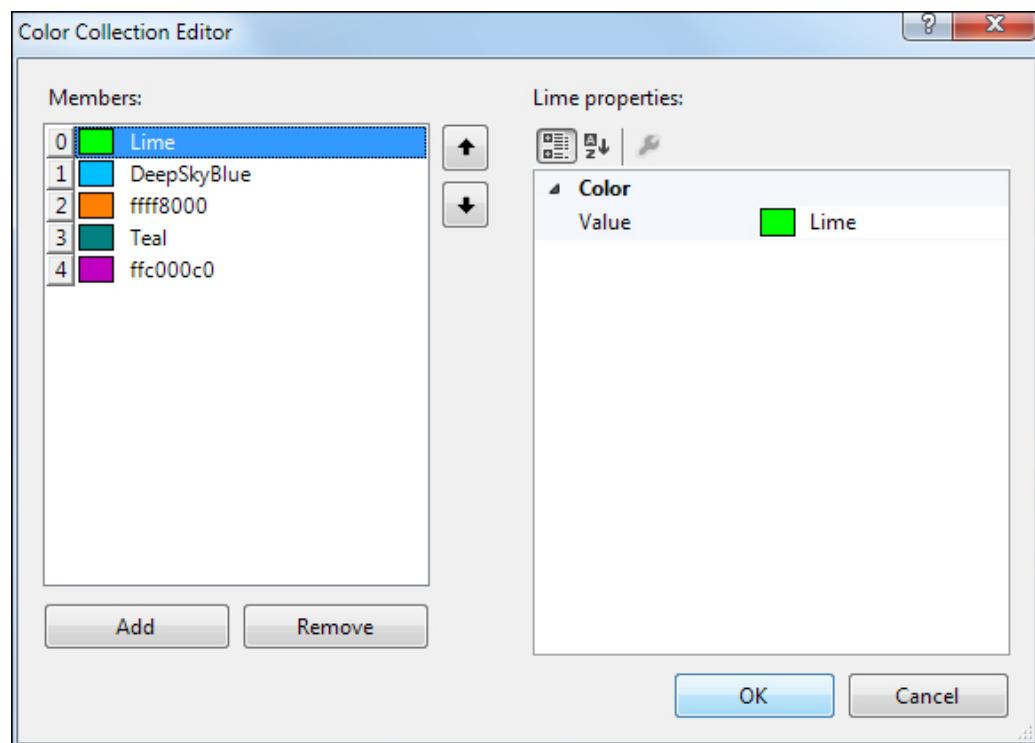
But click the button to the right of **PaletteCustomColors**. This brings up the **Color Collection Editor**. Click the Add button to add a new colour:



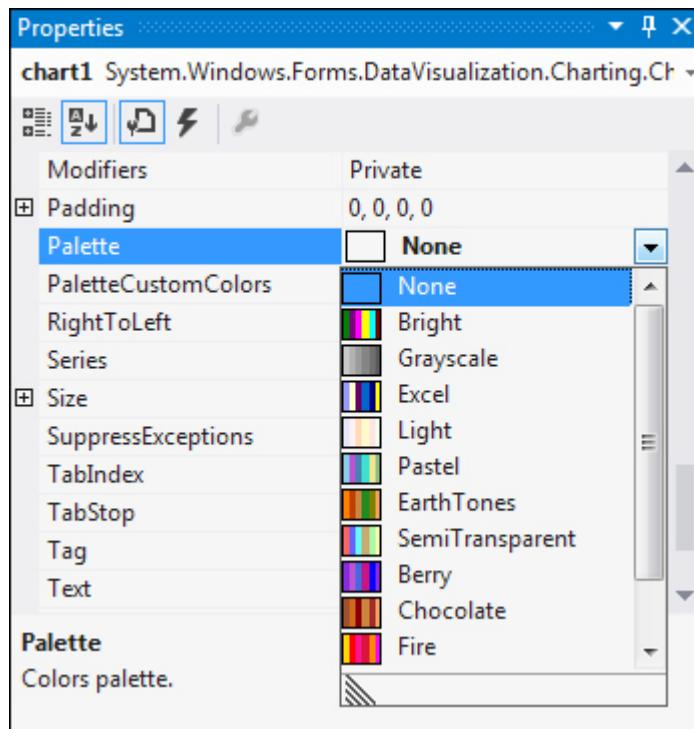
The default value is white, so click the dropdown list and select a new colour:



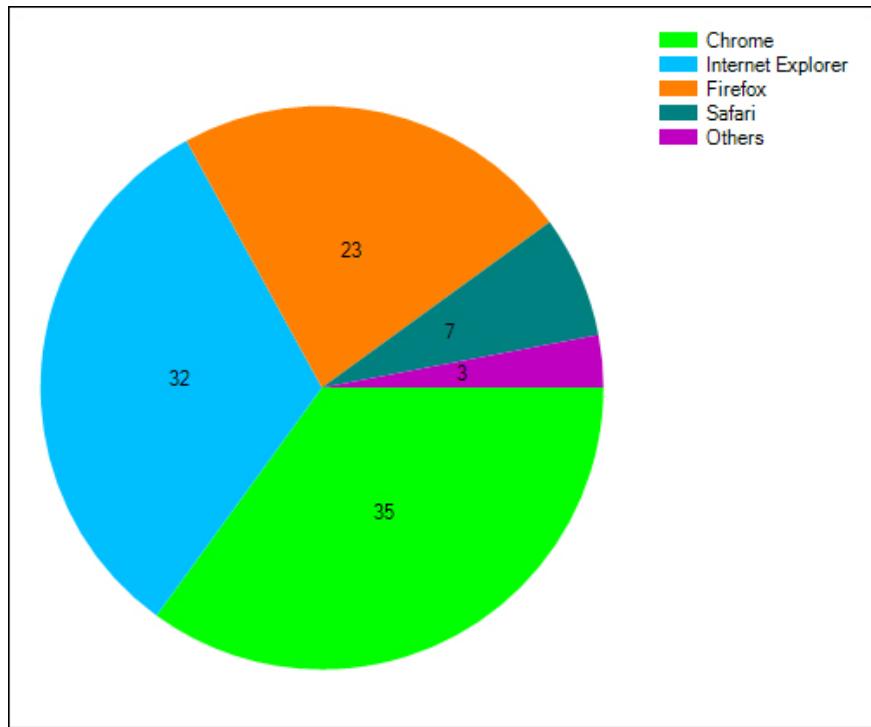
Add four more colours and your Color Collection Editor will look like this:



Click OK to return to Design View. Now have a look at the properties areas. Set the **Palette** property to **None**:

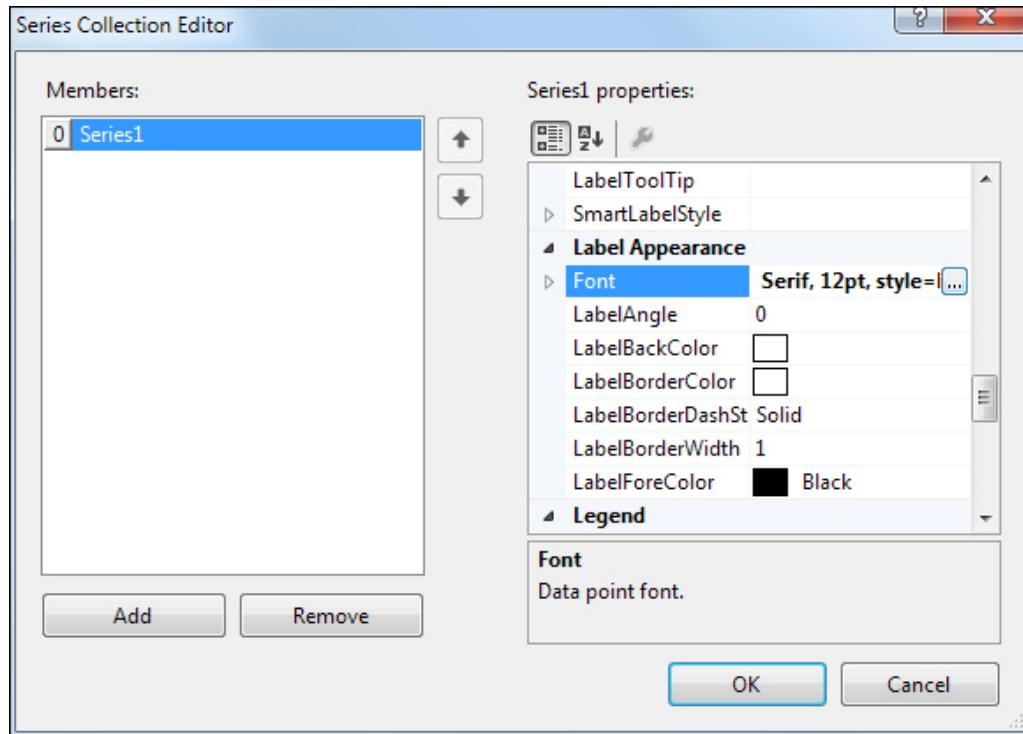


Setting Palette to None will ensure that your own custom colours get used instead. Run your programme and your form will look something like this:



As you can see, we now have the colours we want.

The numbers on each slice of the pie chart look a little dull, though. You can change them via the Series Collection Editor. So bring up the Series Collection Editor like you did before. Locate the **Label Appearance** category, and the Font property. Set a new font style for your labels:



Explore the other properties in the **Label Appearance** category. See what you can come up with for your labels.

Exploding Pie Slices

The slices in our pie chart are all joined together, with no separation at all. If you want to separate the slices (called exploding a point) , you need to do it with code.

So return to the code your form load event.

The slices in the pie are all points, which are in a collection. The points collection in turn is part of the series collection. To get at an individual point, you start like this:

```
chart1.Series["Series1"].Points[0]
```

Each point in the series collection called "Series1" can be access using square brackets. In the code above, we're accessing the first point , which is **Points[0]**.

To explode a point you need the **Exploded** property. You then set this to either **True** or **False**. To get at the **Exploded** property you type it between square brackets. Like this:

```
chart1.Series["Series1"].Points[0]["Exploded"] = "True";
```

The word "Exploded" goes between quote marks. The value True or False goes after an equal sign, again in double quotes. Notice where all the square brackets are in the line above.

To explode all the points, then, we can use a for loop. Add the following to your code:

```
for (int i = 0; i < pie_points.Length; i++)
{
    chart1.Series["Series1"].Points[i]["Exploded"] = "True";
}
```

This loop goes round from 0 to less than the length of the **pie_points** array. In between the curly brackets of the loop we're setting each Exploded property to true. We're using the variable **i** to get at each point in the collection.

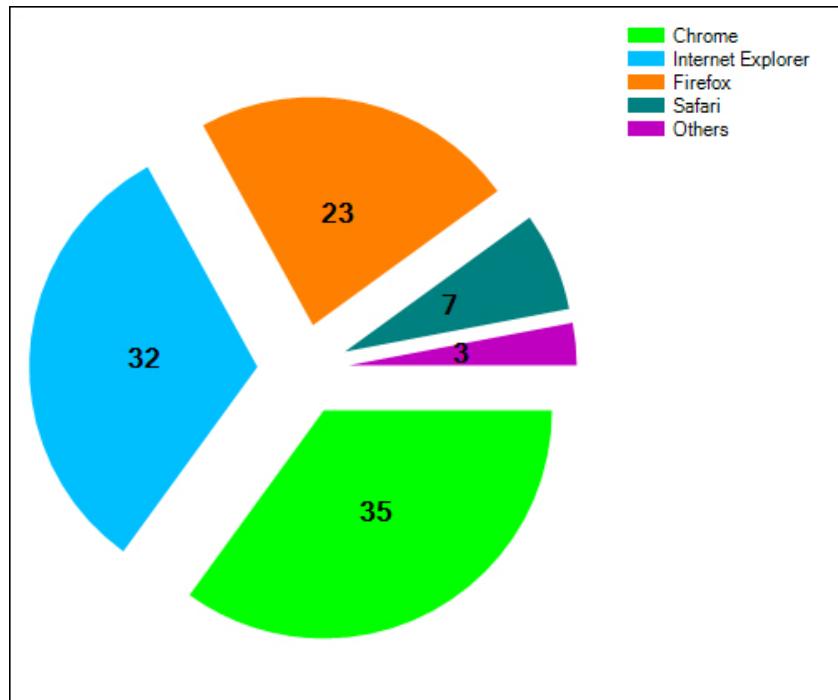
Your Form Load event should now look like this:

```
private void Form1_Load(object sender, EventArgs e)
{
    double[] pie_points = new double[] { 35, 32, 23, 7, 3 };
    string[] series_text = new string[] { "Chrome", "Internet Explorer",
                                         "Firefox", "Safari", "Others" };

    chart1.Series["Series1"].Points.DataBindXY(series_text, pie_points);

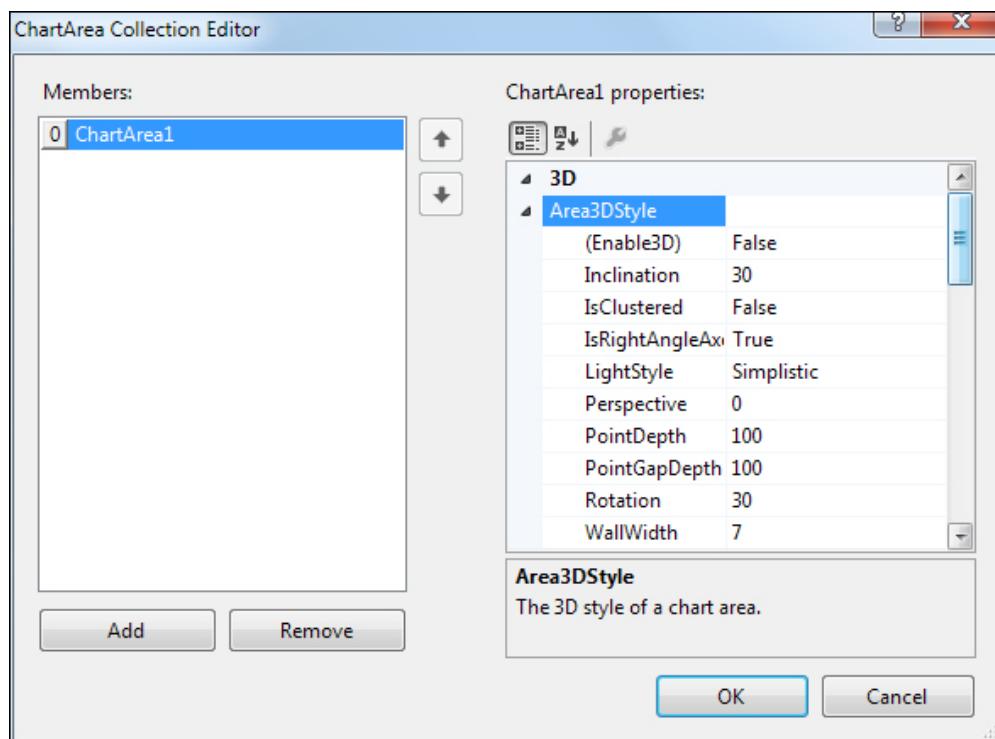
    for (int i = 0; i < pie_points.Length; i++)
    {
        chart1.Series["Series1"].Points[i]["Exploded"] = "True";
    }
}
```

Run your programme and you should find that each slice in your pie chart is now exploded:

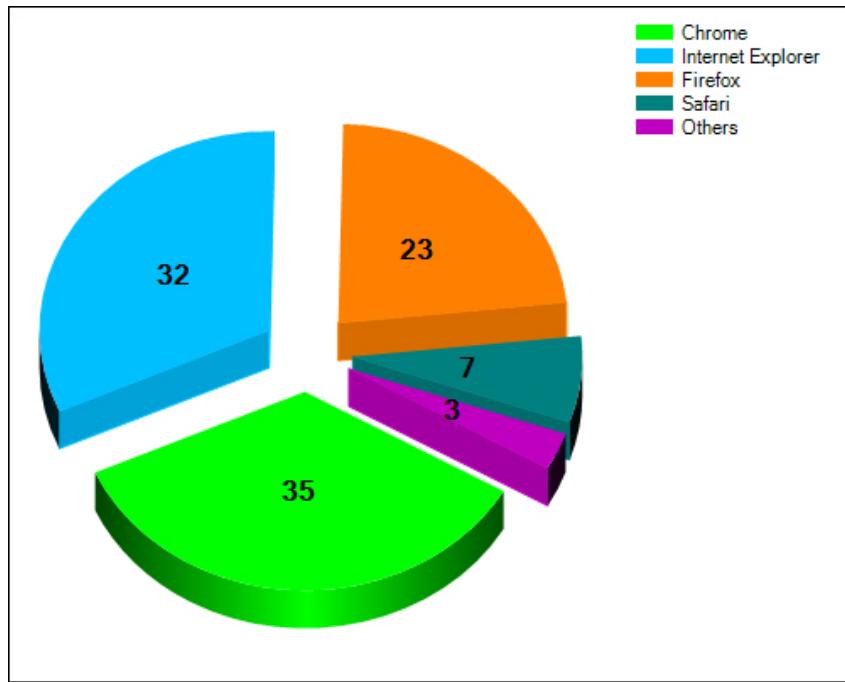


3D charts

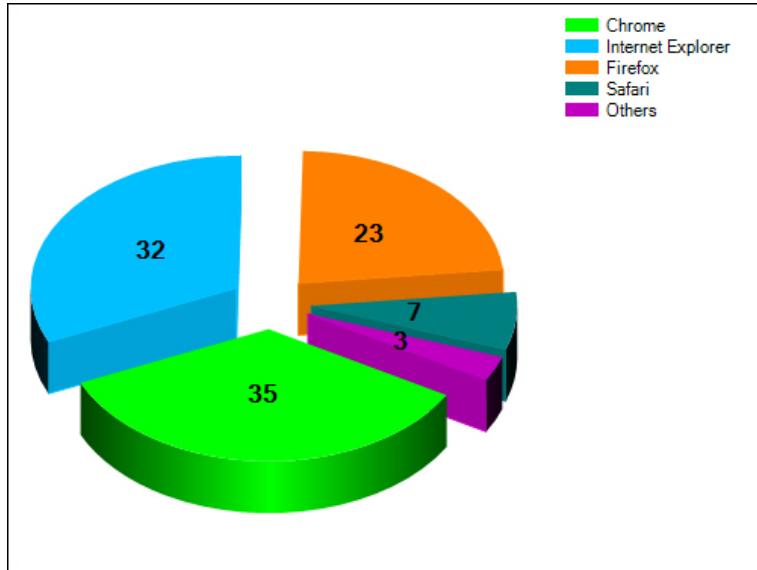
You can get add 3D look to your charts quite easily. In Design View, click your chart to select it. In the properties area on the right locate the **ChartAreas** property. Click the button to the right of **Collections** to bring up the **ChartArea Collection Editor**. Locate the **3D** category at the top and expand the **Area3DStyle** item:



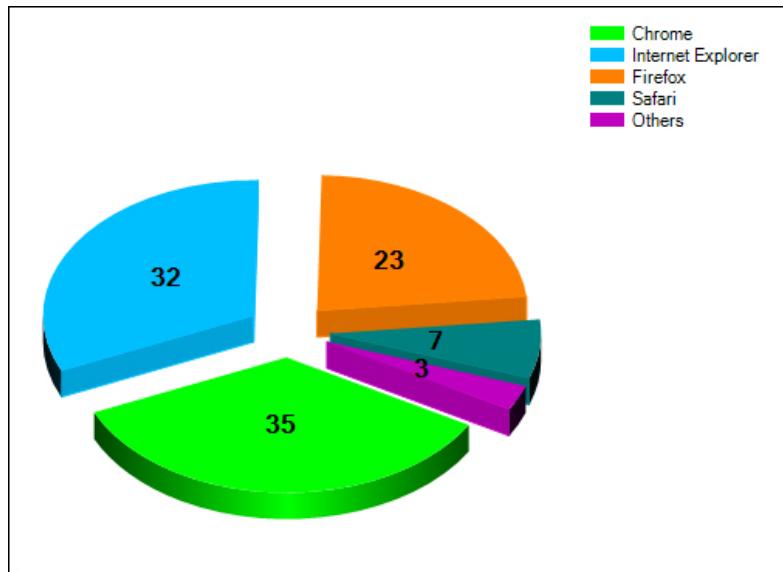
There's quite a lot of 3D options to explore. Start by setting **Enable3D** to **True**. If you just left it at this your chart would be as follows:



Setting the **Inclination** to a value of, say, 50 would get you a chart that looks like this:



If you want to flatter 3D chart, set the **PointDepth** property lower. The chart below has a point depth of 50:

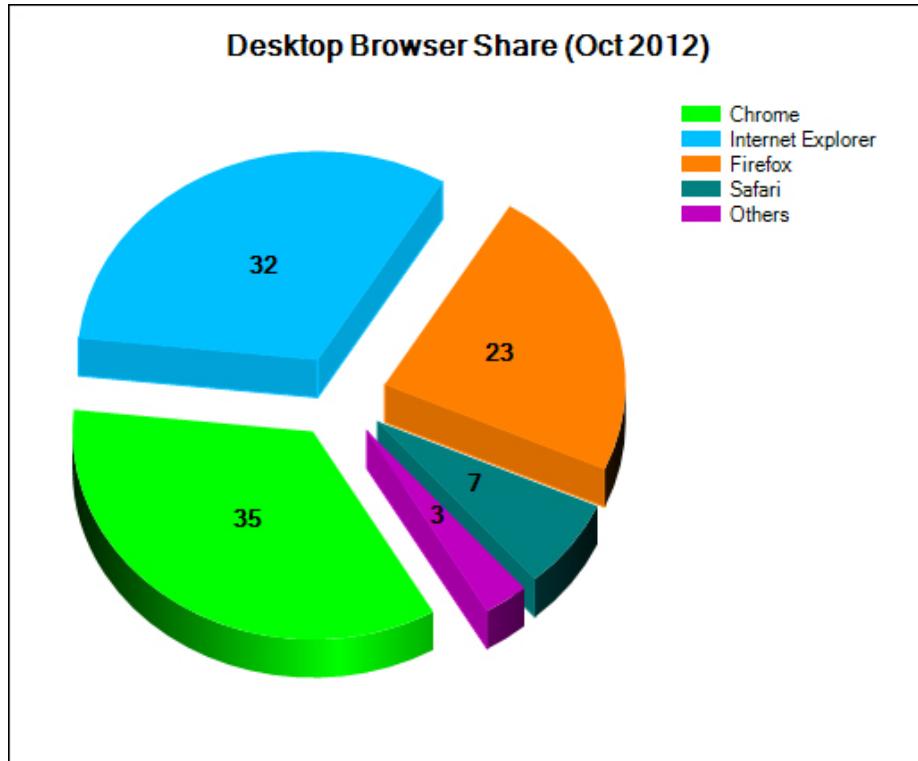


Play around with the other 3D values to see what they do. What does setting the Rotation value to 70 do, for example?

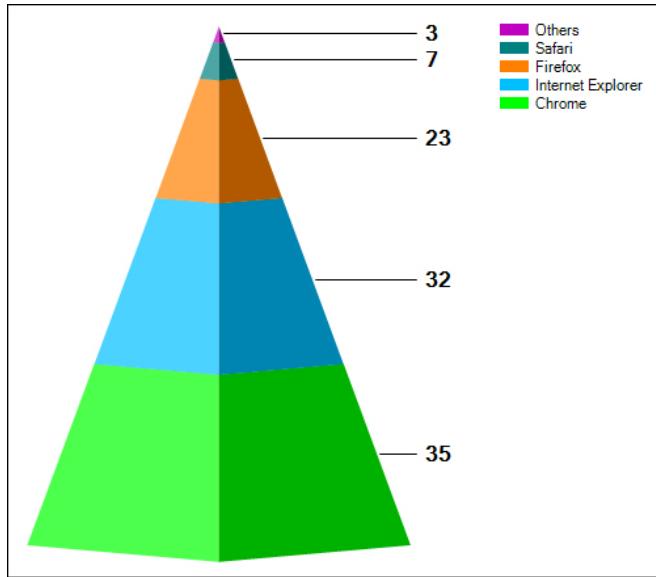
Exercise

To finish off your chart, add a border (in Design View set **BorderLineDashStyle** to **Solid**, then set a **BorderlineColor** and a **BorderLineWidth**). And set a Title, as well (via the **Titles Collection Editor** and the **Title** property). You've already done both of these previously.

Your finished pie chart should look something like ours below:



And that's it for charts. There's a whole lot more to cover, and a whole lot of chart types we haven't touched on. For example, you can easily turn your pie chart into a pyramid through the **Series Collection Editor**. It would then look like this:

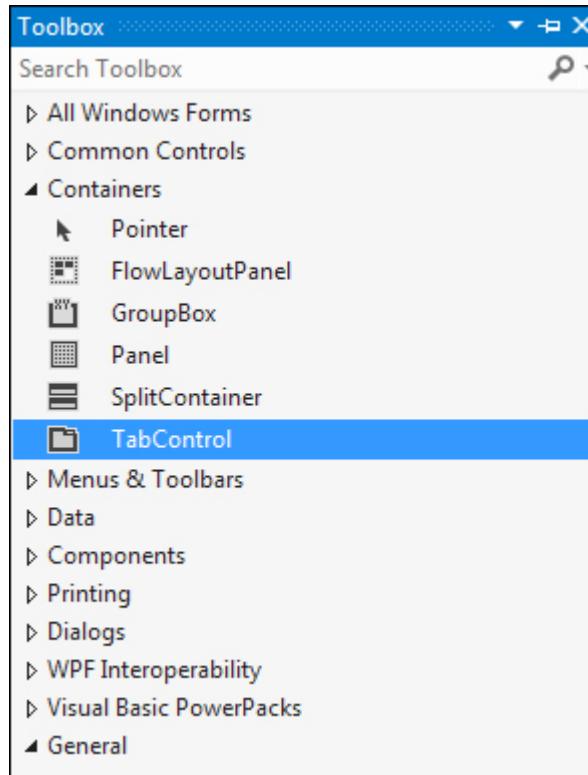


Give it a try yourself!

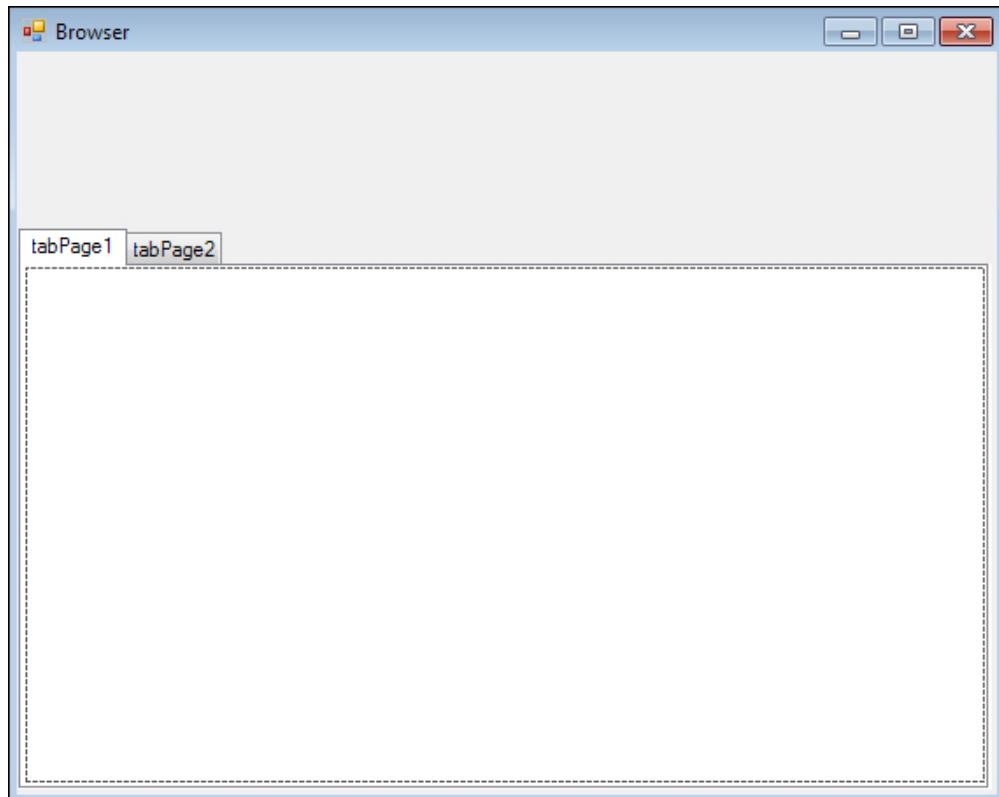
Create Your Own Browser

In this section, we'll show you how to build your own custom browser in C# .NET. We'll only use three control: a SplitContainer, A TreeView and, of course, a WebBrowser control. It's surprisingly easy!

Create a new project for this. Make your new Form nice and big. In the toolbox, locate the **TabControl**, which is under the **Containers** heading:

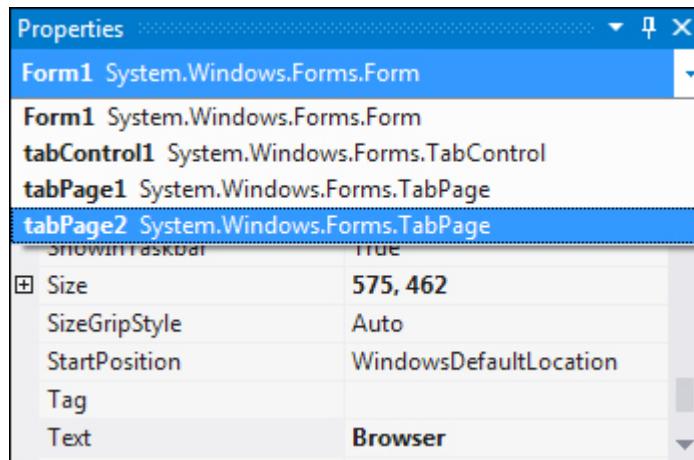


Add one to your form and resize it. Your form should then look like this:



The default TabControl gives you two tabs. We only want one, though.

To delete a tab, you can select it from the properties area on the right. Click the arrow on the drop down box. Locate **tabPage2** and select it:

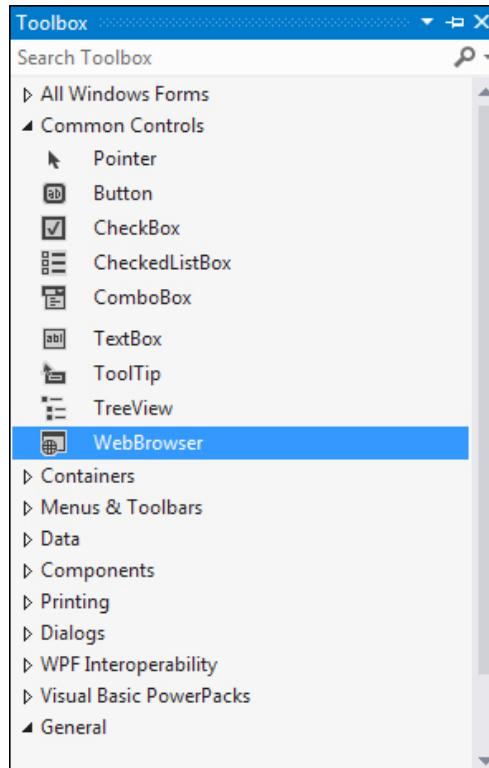


Now right click anywhere on tabPage2 except the tab header (where the tabPage2 text is). From the menu that appears, select the **Delete** option. If your whole TabControl disappears, click **Edit > Undo** from the menu bars at the top of the C# .NET software.

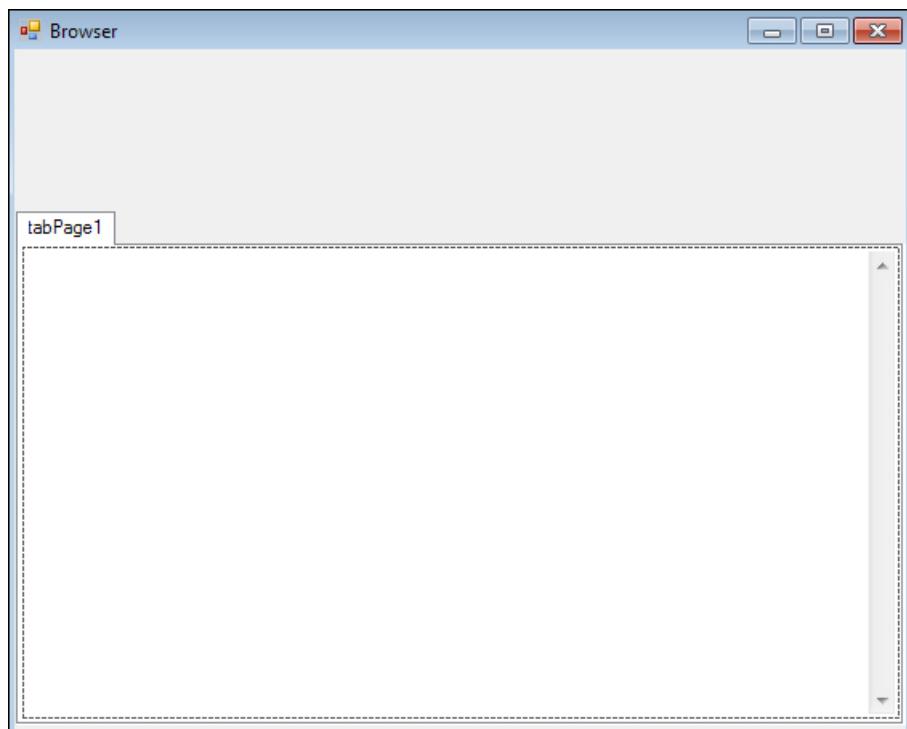
You should now have a TabControl with just one tab. Select this tab from the drop down box in the properties window. We'll add a browser to it.

The WebBrowser Control

Adding a Browser to a Tab is quite easy. Locate the **WebBrowser** control in the toolbox, in the Common Controls category (we've chopped a few controls out, in the image below):

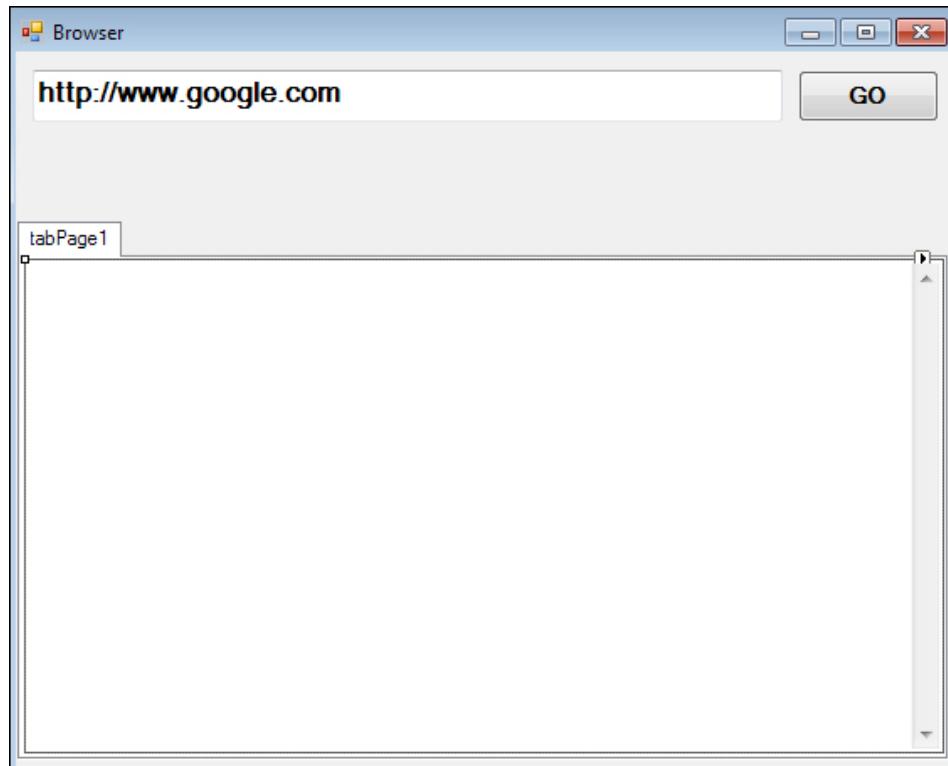


Click the control once to select it. Now draw out a browser in your TabPage1. It should fill the whole tab, and look like this:



The web browser you have just added is an instance of Internet Explorer. It will take the same settings as those from the Internet Options dialogue box in your control panel. So, for example, if you have scripting turned on in Internet Options, it will still be turned on in the WebBrowser control you have just added to the tab.

To see if it works as a browser, though, add a text box and button to your form. Change the Name of the text box to **txtAddress**, and type a web address for the Text property. Change the Name property of the button to **btnGo**, and the Text to **GO**. Aim for something like this:



Now double click your new button to get at the code stub.

The WebBrowser control has a method called **Navigate**. You use it to navigate to a web page that you specify. So add these two lines to your button code:

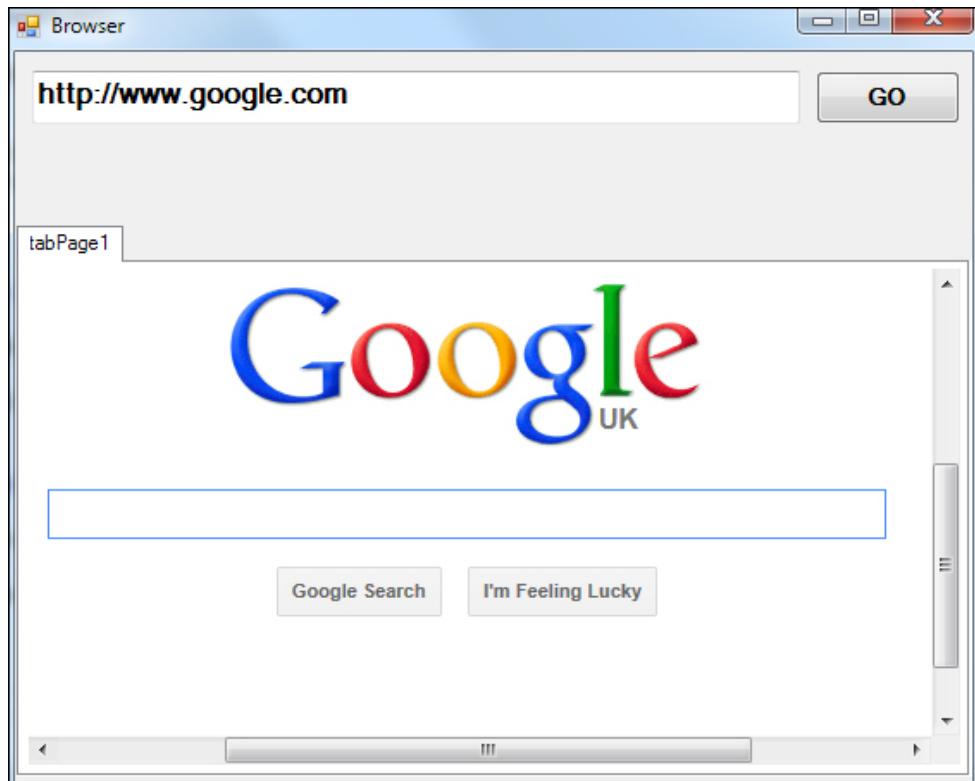
```
string WebPage = txtAddress.Text.Trim();
webBrowser1.Navigate(WebPage);
```

We're just getting the web page address from the text box. You would need to do some error checking here, though, testing for things like blank strings and valid web addresses.

The page you want to navigate to goes between the round brackets of the **Navigate** method. And that's it!

Test it out. Run your programme and click your button. You should find that the web page that you typed in the text box appears in the web browser that you

placed on TabPage1. If it doesn't, make sure that you're online and that your firewall is not blocking your C# .NET software. Here's what ours looks like:



Navigation Buttons

We'll add some navigation buttons, now. The buttons we'll add will allow us to go back one page, move forward one page, go to the home page, cancel the page loading, and refresh the page. Instead of having text on our buttons, we'll have images.

For the button images, we'll have an ImageList that will allow us to select a picture for all the buttons on the form.

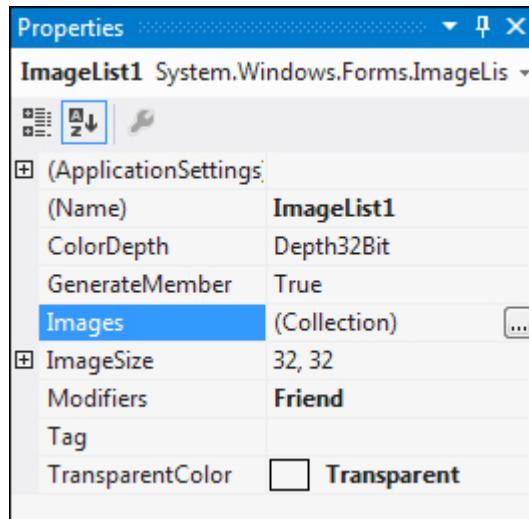
You'll need some images for your buttons. We're using the free icon set from this web page:

<http://www.woothemes.com/2009/09/woofunction-178-amazing-web-design-icons/>

The images are all PNG files. If the above link no longer works, try Googling the term "free icons" (with the quote marks). Or search your hard drive for suitable images. You can search for files that end in, say, GIF by entering *.gif in the Windows search box. Go for an image size no bigger than 64 pixels high by 64 pixels wide. Image types supported by the ImageList control are JPEG, GIF, BMP, PNG and ICO.

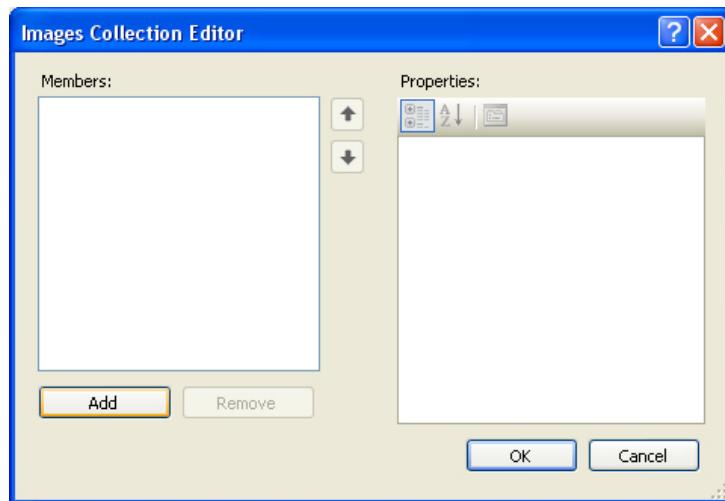
So, add an ImageList control to your project (under the **Components** category in the toolbox).

The ImageList has a **ColorDepth** property that we need to change. Our images are in the PNG-24 format, where the 24 stands for the number of bits. You'd think, then, that we'd need to set the ColorDepth to 24Bit. However, they look awful at this Depth. When we switch to 32Bit, they look fine! But if your images look awful, change the ColorDepth property.

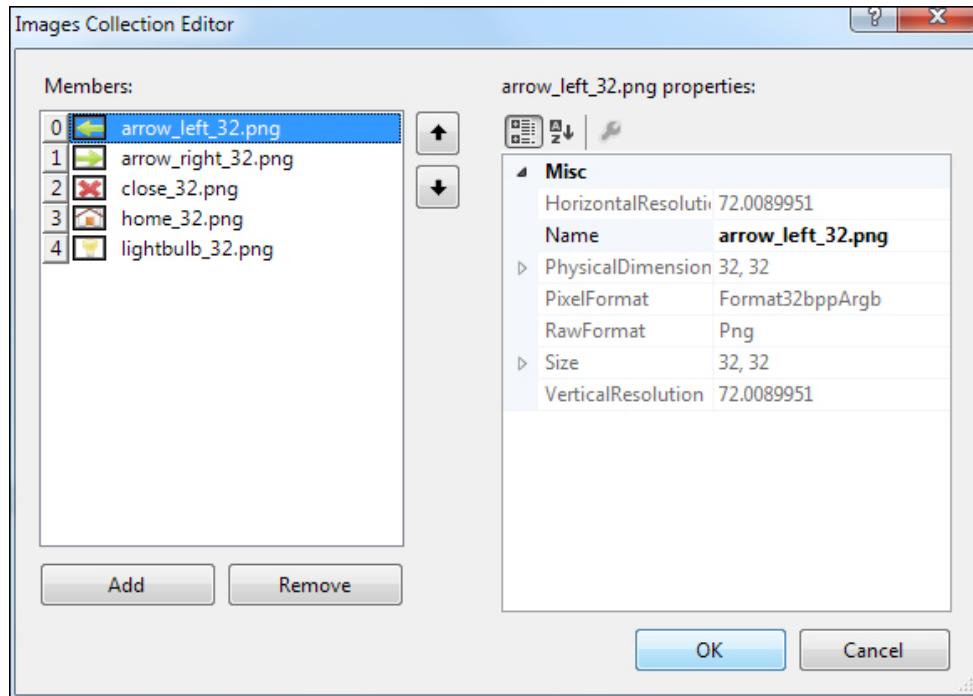


To add images to your image list, click the **Images** button in the Properties area, just to the right of **Collection**, in the image above.

When you click the small button, you'll see the following dialogue box appear:

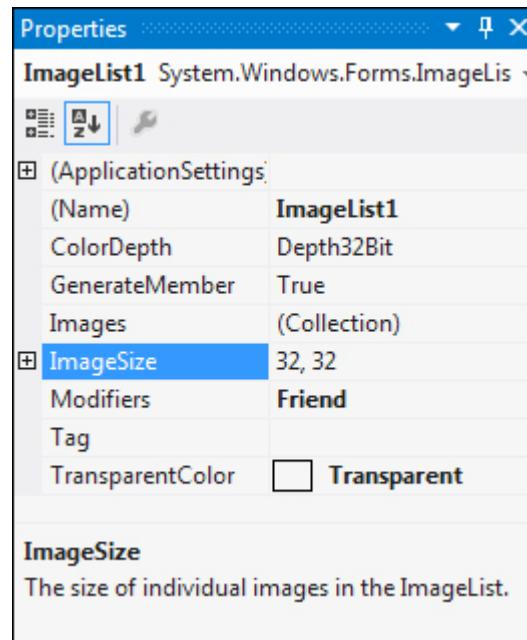


Click the Add button to add some image to your ImageList. We've added five to ours, in the image below:



(We're using the light bulb as a refresh icon.) Click OK when you're done.

Our images are 32 pixels high by 32 pixels wide. We can change to that size in the ImageList properties area. The default is to have the images 16 by 16. Type the new size into the **ImageSize** property, if you need to:

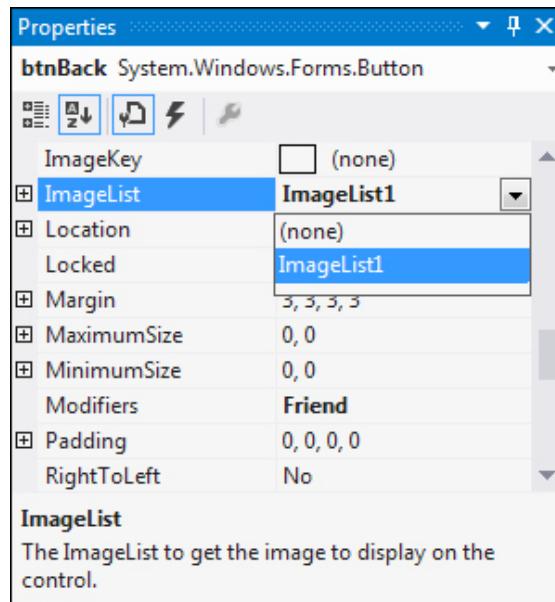


Graphic Buttons

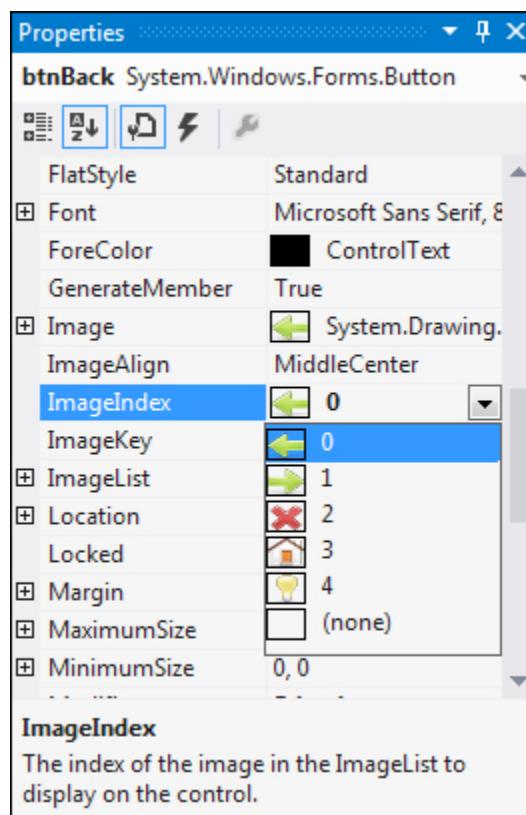
Adding graphics to your buttons is quite easy. Add a button to your form, using the toolbox area on the left. Change the Name property to **btnBack**. Delete the

default text from the Text property of the button, leaving it blank. Resize your button to a suitable size.

To add an image from your ImageList, locate the ImageList property. From the dropdown box, select the name of your ImageList for **btnBack**:



Now locate the **ImageIndex** property. Click the dropdown box and you'll see a list of all your available images. Select the image you're going to use for your Back button. We're using a left-pointing green arrow for ours, which has the ImageIndex of 0:



Once you've selected an image, take a look at your button. It should have your picture on it:



The code for the back button is quite simple. So double click your button to get at the coding window. Add the following:

```
if (webBrowser1.CanGoBack)
{
    webBrowser1.GoBack();
}
```

We're using an if statement first to check whether there is a page to go back to. This is done with the **CanGoBack** property of the WebBrowser object.

If there is a page to go back to, we then use the built-in **GoBack** function. This will force the browser to go back to the page you were previously looking at.

Exercise

Add four more buttons to your Form. Set up the following properties for your buttons:

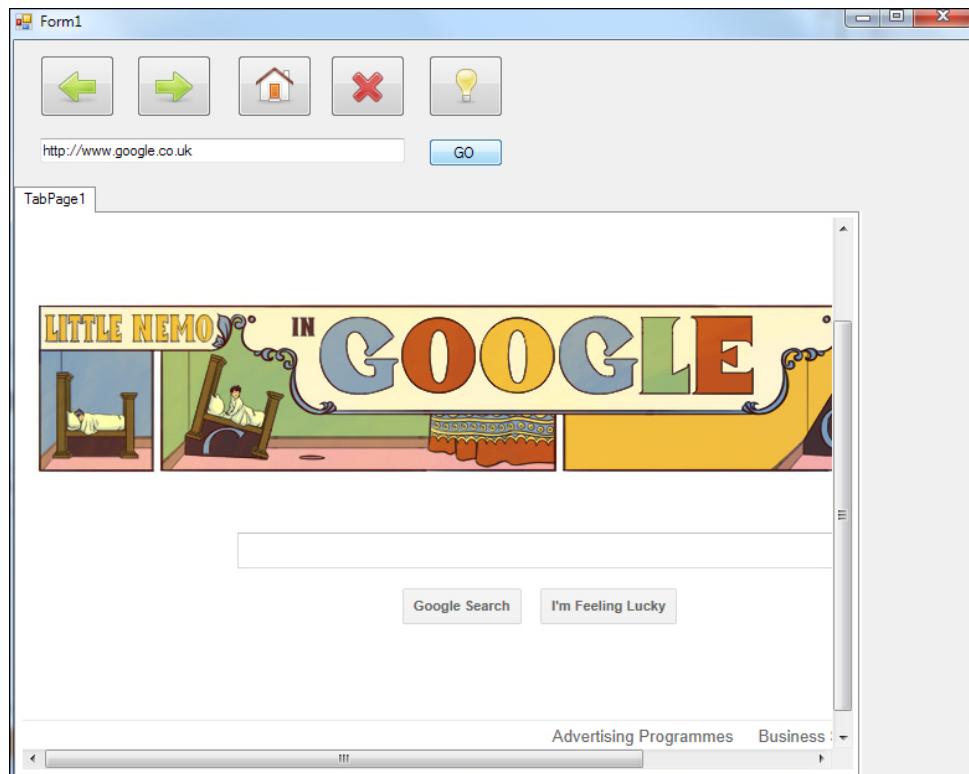
Name: btnForward
Image: An image of your choice

Name: btnHome
Image: An image of your choice

Name: btnStop
Image: An image of your choice

Name: btnRefresh
Image: An image of your choice

When you're done, your form may look something like ours:



Test out your programme, though. Navigate to a web page like Google. Enter something in the search page, just so as to bring up a second page. Now click your Back button. You should be taken back to the previous page.

Exercise

Below, you'll find four pieces of code. Add the right code to the appropriate button:

```
if (webBrowser1.CanGoForward)
{
    webBrowser1.GoForward();
}
```

```
webBrowser1.Stop();
```

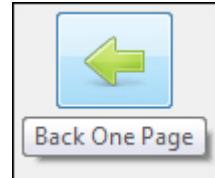
```
webBrowser1.GoHome();
```

```
webBrowser1.Refresh();
```

So you're adding code to each of your remaining four buttons. When you're done, all four buttons should work, when you run your programme.

Adding ToolTips

When you hold your mouse over a button, it would be nice to have a ToolTip display. The ToolTip will tell the user what the button does. Sadly, adding a ToolTip is not quite as straightforward as it should be. But what we want to add looks like this (the yellow box in the image below):



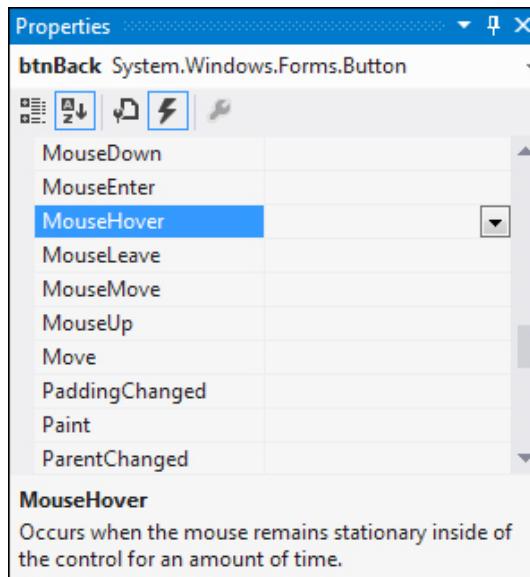
To get ToolTips, you need to create a ToolTip object. You then pass it the name of the form object you want the ToolTip for, and the text you want to display.

The ToolTip control can be added using the toolbox. It's under the **Common Controls** category.

Double click to add one to your project, and you'll see it appear at the bottom of the screen, rather than on the form.

You want the text of the ToolTip to appear when the mouse is over a button. Buttons have a Hover event, so we can use that for the code.

To get to the Hover event, click on a button to select it. In the properties area on the right, click the lightning bolt icon to see a list of event. Locate the **MouseHover** event:

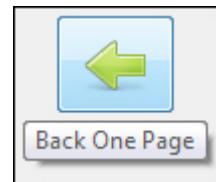


Double click where it says **MouseHover**, and it will open up the code stub. Now enter the following code:

```
toolTip1.SetToolTip( btnBack, "Back One Page" );
```

So the ToolTip has an in-built method called **SetToolTip**. In between the round brackets of **SetToolTip** you need two things: the name of the object you want the ToolTip for, and the Text to go in the yellow box. We called our button **btnBack**, so that's we put as the first argument. The text comes after a comma, and in quote marks.

Run your programme and test it out. Now hold your mouse over your button and you should see your ToolTip:



Exercise

Add ToolTips for your other four buttons. You can have any text you want for your ToolTips.

Exercise

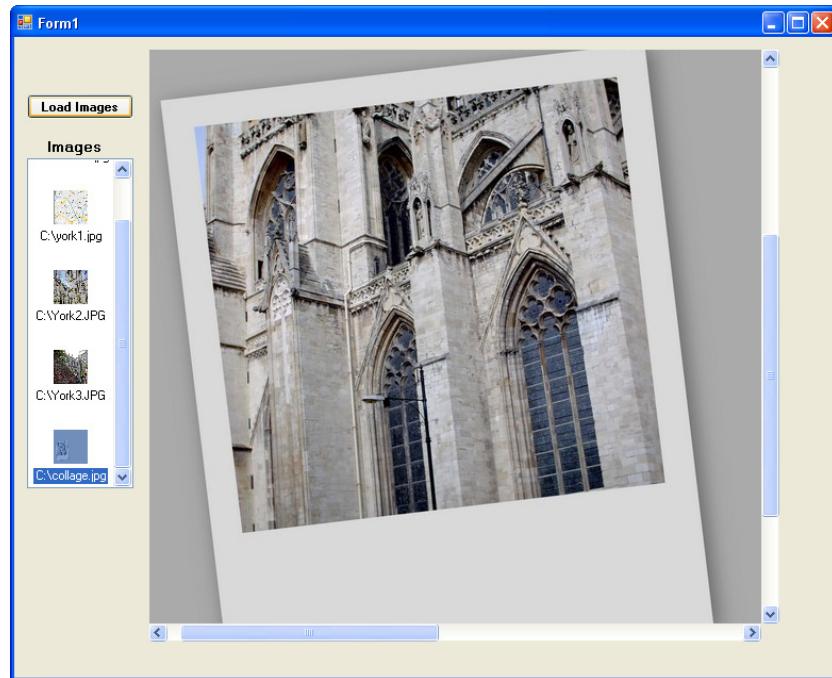
If you wanted to, you could have a Rollover effect for your buttons. This is when the image on the button changes when the mouse hovers over it. When you take your mouse away, it will go back to the original image. The only thing you need for your **MouseHover** event is this:

```
btnBack.Image = imageList1.Images[2];
```

What you're doing here is setting the Image property of the button to one of the images from your Image list, `Images[2]` in the code above. You can get the numbers from the dropdown list you saw earlier, for the **ImageIndex** property of the button. Use the **MouseLeave** event to reset the image to the original.

Picture Viewer – A Project

In this section, you'll create your very own Picture Viewer. It will look like this:



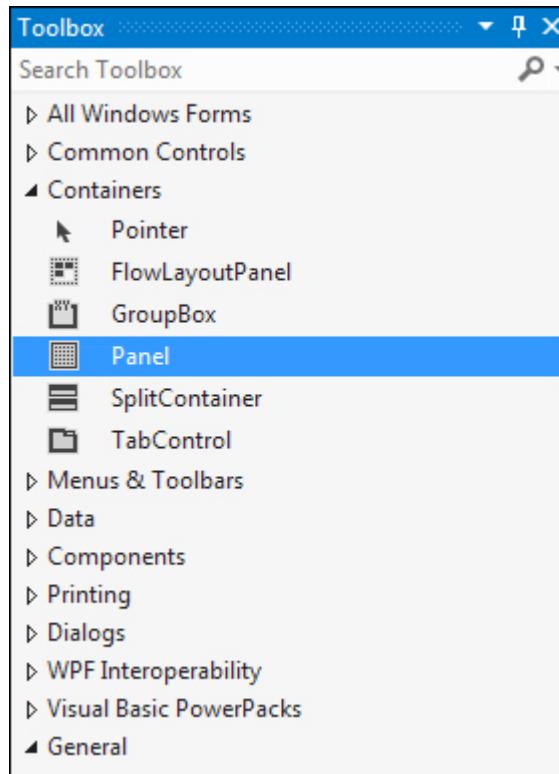
Down the left hand side, we have a ListView control. This allows us to display thumbnails of our chosen images. When you click an image in the ListView, the full size image appears on the right. If the image is too big for the form, scroll bars will appear. Our programme also has a rudimentary zoom ability. When you right click the big image, the picture becomes smaller. When you left click the image, it will become bigger. You'll learn to use the following controls:

ListView
ImageList
Panels
Picture Box
Open File Dialogue Box

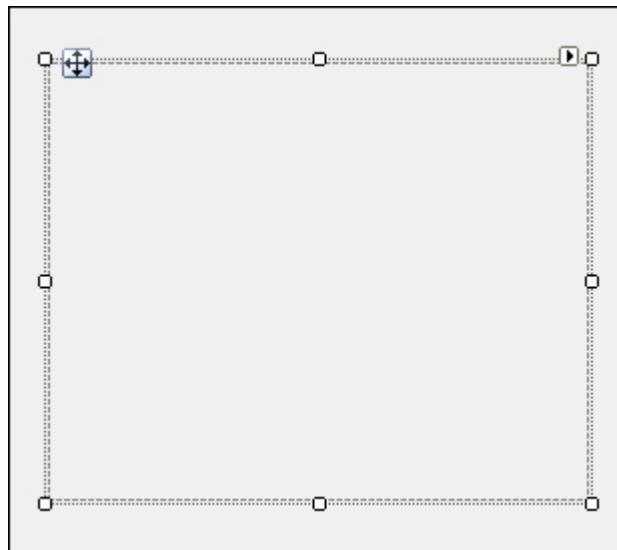
Let's make a start.

Adding the Controls to the Form

Start a new project. Make your Form nice and big. We made ours 950 wide by 800 high. Add a **Panel** control to the form. This can be found in the toolbox, in the **Containers** section:



Draw the Panel any size you like. When you do, you'll see that it has a Move icon top left:



With the Panel selected, set the following properties for it:

| | |
|--------------------|--------------------------|
| Location: | 160, 10 |
| Size: | 750, 720 |
| Anchor: | Top, Bottom, Left, Right |
| AutoScroll: | True |

The Panel will hold a PictureBox control. The reason we're adding a Panel is because the PictureBox doesn't have scrollbars of its own. We can dock a

PictureBox to a Panel. When you make the PictureBox's image bigger than the Panel, the Panel will add the scrollbars automatically.

So add a PictureBox to your Panel. (The PictureBox is under the **Common** category in the Toolbox.) Just click the PictureBox once. Now draw out a PictureBox to fill your Panel.

Set the following Properties for your PictureBox:

| | |
|------------------|----------|
| Dock: | Top |
| SizeMode: | AutoSize |

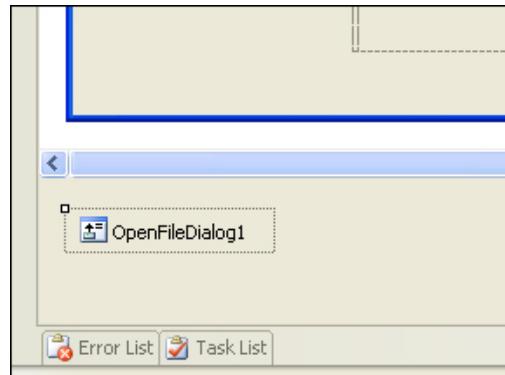
Your Picture Box should now be entirely inside of your Panel.

Now add a button to your Form. Change the Name property to **btnLoadImages**, and set the Text to **Load Images**. Position the button somewhere in the top left of the form.

Just below the button, you can add a Label with the Text property of **Images**.

Add a ListView control to your form. The ListView is also under the **Common** category in the Toolbox. Resize your ListView control, and place it under your button and Label. (We made our Size 120, 300.) Leave the other properties on their defaults.

The next thing to add is a **OpenFileDialog** control. This can be found under the **Dialog** category of the Toolbox. Simply double click to add one to your project. You'll see it appear at the bottom of the project window:

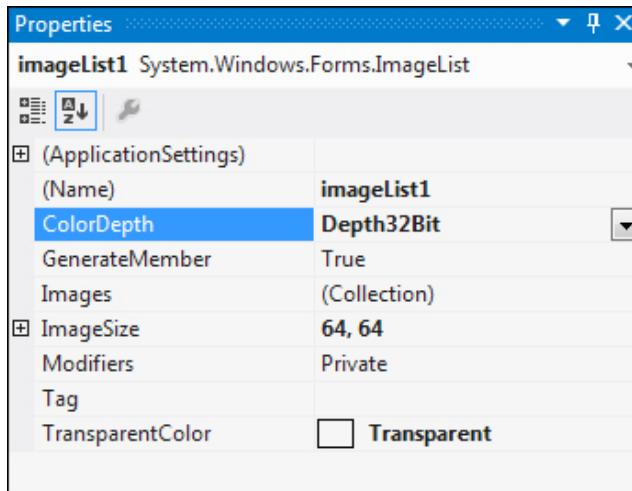


With the **OpenFileDialog** control selected, have a look at the Properties area on the right. Set the **Name** property to **oFD1**, and the **MultiSelect** property to **True**.

Finally, add an ImageList control. This can be found under the **Components** category of the Toolbox:

Double click ImageList to add one to your project. Again, it will get added to the bottom of the project Window.

With your ImageList1 control selected, set its **ImageSize** property to 64, 64. Set the **ColorDepth** to **Depth32Bit**:



The ImageList control, as its name suggests, holds a list of images. We'll use these images as the thumbnail images for the ListView.

Now that we have all our controls in place, we can start the coding.

Selecting Images

The first we'll do is to launch the Open File Dialogue box. You've done this before, in a previous section.

Double click your button to get at the code stub. The first two lines of code to add will clear the ImageList of its images, and clear the ListView. Add these two lines, then:

```
imageList1.Images.Clear();
listView1.Clear();
```

So we're just issuing the **Clear** command of the two objects.

To get the Open File Dialogue box to appear, add these lines:

```
oFD1.InitialDirectory = "C:\\";
oFD1.Title = "Open an Image File";
oFD1.Filter = "JPEGS*.jpg|GIFS*.gif";

var ofdResults = oFD1.ShowDialog();

if (ofdResults == System.Windows.Forms.DialogResult.Cancel)
{
    return;
}
```

You should know what these lines do by now, as you've met them before. But we're just setting the initial directory, adding a title to the dialogue box, and then specifying the type of files than can be opened. We have to handle the Cancel button being clicked, which is what the rest of the lines do.

You can try it out, if you like. The dialogue box should appear when you click your button, and you should be able to select more than one file.

As you learned earlier, though, the Open File Dialogue box doesn't actually open files. It just allows you to specify the names of the files you want to open.

Because you want the user to be able to load more than one file into the ListView area, you have to get at all the file names selected. This can be done with a **foreach** Loop. Add the following lines to the ones you already have:

```
try
{
    foreach( string single_file in oFD1.FileNames ) {
        MessageBox.Show(single_file);
    }

}
catch (Exception err)
{
    MessageBox.Show("Error: " + err.Message);
}
```

We've put it in a **try ... catch** block, to trap any errors. The **foreach** loop is this:

```
foreach( string single_file in oFD1.FileNames ) {
    MessageBox.Show(single_file);
}
```

So the OpenFileDialog control has a property called **FileNames**. This gets you a list of all the files that the user selected. By using **foreach** you can loop round and get all the file names (which will include the file paths). The file names will be stored in the **single_file** variable.

Try your programme out again. Click your button, and select a few images on your computer. When you click the Open button, you should see the Message Box display, with all your file names and file paths displayed.

We need all those file names and file paths. We can then pass them to the ImageList control, for the thumbnails. But we'll also need them for the ListView control. To store the file names and file paths, we can use an array. Add the following to your **try ... catch** block, just before the **foreach** loop:

```
int num_of_files = oFD1.FileNames.Length;
string[] aryFilePaths = new string[num_of_files];
```

The **Length** property of **FileNames** gets you how many files were selected. In between the round brackets of **aryFilePaths**, we're using this number to set the size of the array.

Add this third variable, just below the two above:

```
int counter = 0;
```

We can use the counter to access the array. In place of your Message Box, add this line to your **foreach** loop code:

```
aryFilePaths[counter] = single_file;
```

So the file names and paths will now be stored in the array we set up. Increment the counter on the line just below this one:

```
counter++;
```

Your button code should now look like this, though:

```
imageList1.Images.Clear();
listView1.Clear();

oFD1.InitialDirectory = "C:\\\\";
oFD1.Title = "Open an Image File";
oFD1.Filter = "JPEGS|*.jpg|GIFS|*.gif";

var ofdResults = oFD1.ShowDialog();

if (ofdResults == System.Windows.Forms.DialogResult.Cancel)
{
    return;
}

try
{
    int num_of_files = oFD1.FileNames.Length;
    string[] aryFilePaths = new string[num_of_files];
    int counter = 0;

    foreach( string single_file in oFD1.FileNames) {
        aryFilePaths[counter] = single_file;
        counter++;
    }
}
catch (Exception err)
{
    MessageBox.Show("Error: " + err.Message);
}
```

Adding Images to an Images List

An ImageList control is something you need to add images to. You do this with the **Add** command. We can add all the images that the user selected. This can be done in your **foreach** loop:

```
foreach( string single_file in oFD1.FileNames ) {

    aryFilePaths[counter] = single_file;
    imageList1.Images.Add( Image.FromFile( single_file ) );
    counter++;
}
```

The new line to add is in bold text, above. The **imageList1** object has an **Images** collection. After a dot, use the word **Add**. In between the round brackets of **Add**, you specify the image that you want to add to your ImageList. For us, this was an image that the user selected, which is stored in the **single_file** variable.

You can't just use the file path, however. It has to be an Image. You can turn the file path into an image with the **Image** object. This has a **FromFile** function. In between the round brackets of **FromFile**, you specify a file path:

Image.FromFile(single_file)

So we're creating an image from the file path, and then adding it to the ImageList collection.

When the **foreach** loop is complete, you'll then have an ImageList filled with Images. The size of each image will be 64 by 64, which is the size we set earlier.

After the **foreach** loop, you then need to attach the ImageList to the ListView control. There's only one line of code to add for this:

listView1.LargeImageList = imageList1;

We're using the **LargeImageList** property of the ListView control. This, not surprisingly, will mean that the ListView can display large images. The large images are all in the ImageList.

Adding Images and File Paths to the ListView

So that the user can see the images, we still need to **Add** them as **Items** to the ListView. The previous line just attached the ImageList to the ListView object. But it won't do anything useful with them.

We need to add a **For** loop, now. We'll loop round adding new items to the ListView. The items we'll add are the images to display, as well as some text

under the image. The text will be the file path of the image. (We'll be using this later.)

So add the following **for** loop to your code:

```
for (int i = 0; i < counter; i++)
{
    listView1.Items.Add(aryFilePaths[i], i);
}
```

The loop goes from 0 to less than the value of **counter**. (The value of the counter variable is the same as the number of images in the array we set up.)

Inside the loop is where we add the Items to the ListView:

listView1.Items.Add()

You have a number of different options to choose from between the round brackets of **Add**. But remember what we're doing here: Adding **Items** to the ListView control.

The first thing we want to add is the Text that goes under the image:

aryFilePaths[i]

The second thing to add is the thumbnail image. This is stored in the ImageList, which the ListView now knows about. But the image we attached to the ListView are stored in a collection. You can access this collection either as a Key or as a number. The number is the position in the ImageList:

Add(aryFilePaths[i], i);

We're using the loop variable, here, which will start at 0. The first time round the loop, this will attach **image 0** in the ImageList collection to the ListView.

The full code for your button should now look like ours on the next page:

```
imageList1.Images.Clear();
listView1.Clear();

oFD1.InitialDirectory = "C:\\";
oFD1.Title = "Open an Image File";
oFD1.Filter = "JPEGs|*.jpg|GIFS|*.gif";

var ofdResults = oFD1.ShowDialog();

if (ofdResults == System.Windows.Forms.DialogResult.Cancel)
{
    return;
}

try
{
    int num_of_files = oFD1.FileNames.Length;
    string[] aryFilePaths = new string[num_of_files];
    int counter = 0;

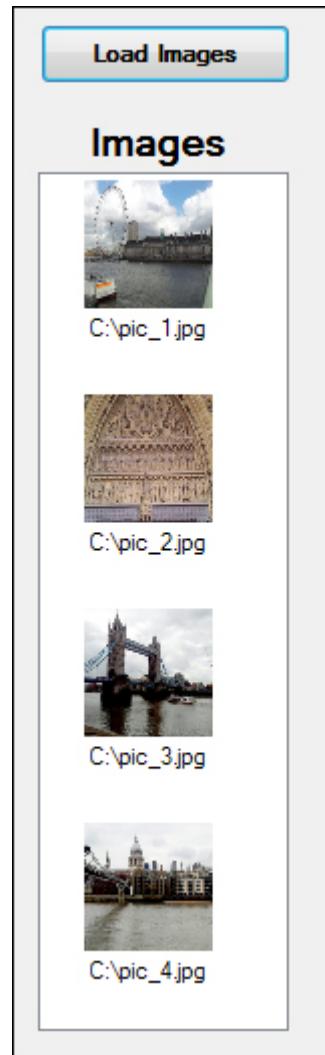
    foreach( string single_file in oFD1.FileNames) {
        aryFilePaths[counter] = single_file;
        imageList1.Images.Add(Image.FromFile(single_file));
        counter++;
    }

    listView1.LargeImageList = imageList1;

    for (int i = 0; i < counter; i++)
    {
        listView1.Items.Add(aryFilePaths[i], i);
    }
}

catch (Exception err)
{
    MessageBox.Show("Error: " + err.Message);
}
```

Try it out. Run your programme and click your button. Select a few images to open. When you click the Open button on your Open File Dialogue box, your ListView control should look something like ours below (obviously, your images and file paths will be different):



So we have a thumbnail image, and the file path of the image. We can use that file path for the bigger image.

The Bigger Picture

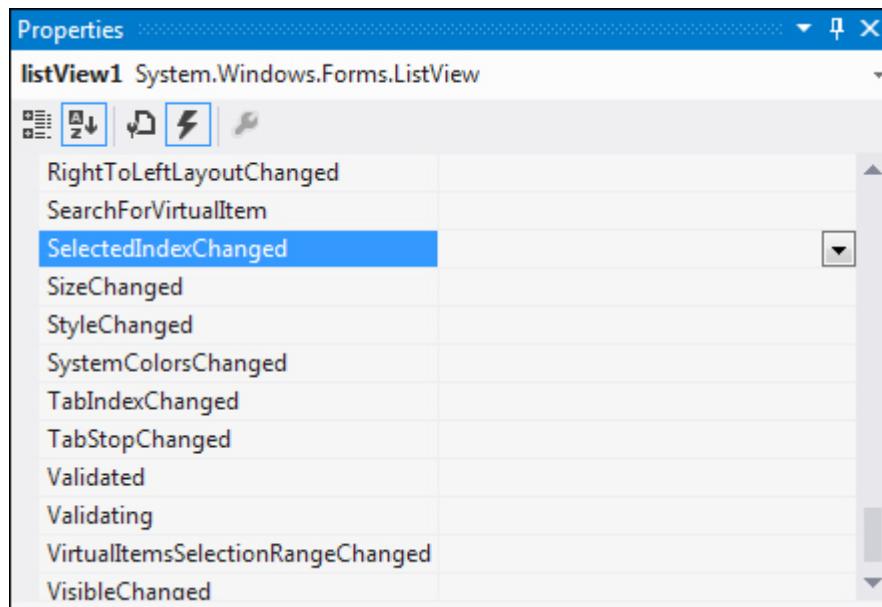
When you click on a thumbnail, you want the picture box on the right to display the full-size version of the image. To do this, you need to get just the file path to the image, which we displayed as text under the thumbnail. This is not quite as straightforward as it should be.

Clicking on a thumbnail fires the **SelectedIndexChanged** event of the ListView control. The Index refers to which complete item you clicked on: the thumbnail, the text, and anything else you have for that particular item. For example, clicking on the first thumbnail in the previous image above would get you a reference to not only the thumbnail image, but also the text "C:\pic_1.jpg". Because it's the first item in the ListView, the index would be 0.

The ListView, however, allows you to select more than one item. You can loop round and get all the items that were selected by the user. For any one item, the options then available in the image above are:

| |
|----------------------|
| Name |
| Size |
| Type |
| Date Modified |
| Dimensions |

So, in Design View, click on your ListView control to select it. In the properties area on the right, click on the lightning symbol to display a list of available events, and locate **SelectedIndexChanged**:



Double click to create the code stub. Now add the following line:

```
string big_filename;
```

This just sets up a variable that we've called **big_filename**. The **For** loop is this:

```
for (int i = 0; i < listView1.SelectedItems.Count; i++)
{
}
```

So we want to go round all the items that were selected. To get at the Text for an item, the code you need inside of your loop is this:

```
big_filename = listView1.SelectedItems[i].Text;
```

The **Text** for us was the file path to the thumbnail. The thumbnail is also the path to the bigger image. We can create a new image from the file path:

```
pictureBox1.Image = Image.FromFile( big_filename );
```

Again, we're using the **Image** object, and the **FromFile** function. In between the round brackets of **FromFile**, we're using our file path. Add the line to your **For** loop.

We only need one more line in the loop. Add this rather curious line:

```
panel1.AutoScrollMinSize = new Size( pictureBox1.Image.Width,
                                pictureBox1.Image.Height);
```

The **AutoScrollMinSize** property of a Panel takes two arguments, a Width and a Height. It's used to get a minimum size for the AutoScroll of the Panel. If you don't include this line then you won't get the scroll bars, or you may just get a vertical scroll bar but no horizontal one.

But the code for your **SelectedIndexChanged** event should look like this:

```
private void listView1_SelectedIndexChanged(object sender, EventArgs e)
{
    string big_filename;

    for (int i = 0; i < listView1.SelectedItems.Count; i++)
    {
        big_filename = listView1.SelectedItems[i].Text;
        pictureBox1.Image = Image.FromFile(big_filename);
        panel1.AutoScrollMinSize = new Size(pictureBox1.Image.Width,
                                pictureBox1.Image.Height);

    }
}
```

Try it out. Run your programme and open some images. Click on a thumbnail in your ListView and you should find that the bigger image appears in the PictureBox on the right.

Zooming in and out

The first thing we'll do is to make the image smaller when the right mouse button is clicked on the image in the PictureBox (zooming out).

What we need to do, though, is to resize the image. We'll use the graphics object to do this, and create new bitmap objects. Tricky stuff!

Go back to Design View and select your PictureBox. Locate the **MouseDown** event in the properties area on the right (click the lightning symbol). Double click the event name to get the code stub. Now enter the following lines of code:

```

private void pictureBox1_MouseDown(object sender, MouseEventArgs e)
{
    if (e.Button == System.Windows.Forms.MouseButtons.Right)
    {

        Bitmap bmp = new Bitmap(pictureBox1.Image);

        Bitmap bmp_new = new Bitmap(Convert.ToInt32(pictureBox1.Image.Width / 2),
                                    Convert.ToInt32(pictureBox1.Image.Height / 2));

        Graphics gr = Graphics.FromImage(bmp_new);
        gr.DrawImage(bmp, 0, 0, bmp_new.Width, bmp_new.Height);
        pictureBox1.Image = bmp_new;
        panel1.AutoScrollMinSize = new Size( pictureBox1.Image.Width,
                                              pictureBox1.Image.Height );
    }
}

```

The if statement just detects if the Right mouse button was clicked. The first line in the if statement is this:

Bitmap bmp = new Bitmap(pictureBox1.Image);

This creates a new Bitmap object, using the image from the PictureBox.

The second line also creates a new Bitmap object:

**Bitmap bmp_new = new
Bitmap(Convert.ToInt32(pictureBox1.Image.Width / 2),
Convert.ToInt32(pictureBox1.Image.Height / 2));**

This time, however, we're just specifying a new size for the bitmap. The new width is this:

Convert.ToInt32(pictureBox1.Image.Width / 2)

And the new height is this:

Convert.ToInt32(pictureBox1.Image.Height / 2)

So we take the Width of the image in the PictureBox and divide by 2. We then need to convert this to an Integer (**Convert.ToInt32**). We do the same for the Height. This will leave us with a blank image, but with the size we need.

The next line is this:

Graphics gr = Graphics.FromImage(bmp_new);

We're creating a graphics object here. The graphics object has a **FromImage** method. In other words, we're setting the new image to be the graphics object. It's still a blank image, though.

The next line is where we draw an image:

```
gr.DrawImage(bmp, 0, 0, bmp_new.Width, bmp_new.Height);
```

The **gr** variable is a graphics object containing an image. The image is blank, however, so we use the **DrawImage** method to draw an image onto it. In between the round brackets of **DrawImage**, we're drawing the original image, but at the new size. (The 0, 0 is the location to start drawing.)

After all this graphics manipulation, the variable we've called **bmp_new** will contain the original image at the new size. This is then used in the next line:

```
pictureBox1.Image = bmp_new;
```

The final line is one you've met before – resize the panel's AutoScroll features.

Try it out. Run your programme and open a big image. Click the thumbnail to load it into your PictureBox. Now right click the PictureBox. Your image should resize.

Exercise

To finish your Picture Viewer, add a zoom in feature. The image in your PictureBox should get bigger when the user LEFT clicks on it. This exercise is a lot easier than you think!

And we'll leave our Picture View there. It's a bit rudimentary, but we hope you learned a lot from it!

Image Editing

In this section, we'll take a look at some of the ways you can edit and manipulate images with C#. You'll learn about image rotation, flipping, inverting, filtering, and how to access the individual pixels in an image. You'll also learn how to save the changes to your images.

You can use your Picture Viewer project from the previous section for this. If you didn't complete the Picture Viewer project then start a new project. Add a PictureBox control to your form. In the properties area on the right, locate the **Image** property. Click the button to the right of the Image property and load an image into your PictureBox control.

Image Rotation

Image rotation is quite easy in C# as there is a method called **RotatFlip**.

Add a button to your form. If you're using Picture Viewer project, resize the PictureBox control and Panel. Or just move them down your form. (You could add a menu system instead, if you're feeling adventurous!) Set the Text property of the button to **Rotate**. Double click your new button to open the code stub for it. Now add the following line:

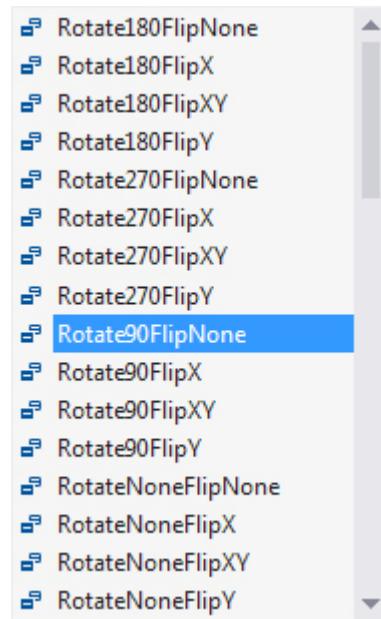
```
Bitmap bmp = new Bitmap( pictureBox1.Image );
```

This line creates a new Bitmap object called **bmp**. There are quite a lot of overloads that the Bitmap constructor can take. One of these overloads just accepts an image object between the round brackets. We're using an existing image between the round brackets of our Bitmap constructor. The image is the one from the PictureBox called **pictureBox1**.

Once we have a Bitmap object to work with, we can use the **RotatFlip** method. Add this line to your code:

```
bmp.RotateFlip( RotateFlipType.Rotate90FlipNone);
```

In between the round brackets of **RotatFlip** you start with a **RotateFlipType**. As soon as you type a dot after **RotateFlipType** you'll see a list of options. These ones:



We've gone for the one that rotates the image 90 degrees but doesn't flip it (you'll see what Flip does soon).

The only other line of code you need here is to assign your new bitmap image to the PictureBox **Image** property:

```
pictureBox1.Image = bmp;
```

When this line executes, your new Bitmap image will replace the image that was in the PictureBox previously.

Before trying it out, you can wrap all three lines of code in an if statement:

```
private void button1_Click(object sender, EventArgs e)
{
    if (pictureBox1.Image != null)
    {
        Bitmap bmp = new Bitmap(pictureBox1.Image);
        bmp.RotateFlip(RotateFlipType.Rotate90FlipNone);
        pictureBox1.Image = bmp;
    }
    else
    {
        MessageBox.Show("No picture");
    }
}
```

What the if statement does is to check if an image has been loaded into the PictureBox. This is done by testing for "Is Not Null".

Try it out. Run your programme and load an image into the PictureBox, if you haven't already. Now click your button. Before rotating, our image looked like this:



After rotating 90 degrees, the image looks like this:



If you keep clicking your button, the image will rotate 90 degrees for each click, until it ends up in the same rotation .

Exercise

Try out some of the other **RotateFlipType** options and see what they do.

Image Flipping

Image flipping is pretty much the same code we had for rotating. The only difference is in selecting one of the other **RotateFlipType** options.

Add a new button to your form. Now add the following code:

```
Bitmap bmp = new Bitmap(pictureBox1.Image);
bmp.RotateFlip(RotateFlipType.RotateNoneFlipX);
pictureBox1.Image = bmp;
```

The new **RotateFlipType** option is highlighted in red above. It's **RotateNoneFlipX**. This option applies no rotation but does apply a Flip on the X axis (the one running left to right).

Try it out. Load an image into your picture box and click you Flip button. You'll see something like this before the button is clicked:



And this after you click your button:



Clone an image

Cloning an image is making a copy of the whole image, or just a part of it. Add a new button to your form and we'll try it out.

The first thing to do is to create a Bitmap of our image from the PictureBox, as before:

```
Bitmap bmp = new Bitmap(pictureBox1.Image);
```

The next you need to do is set up a rectangle object. This is used to specify the size of the cloned area. Add the following to you code:

```
Rectangle rect = new Rectangle(0, 0, 100, 100);
```

The line above sets up a new Rectangle object called **rect**. The first two numbers are the starting point for the rectangle. We're specifying X and Y starting positions of 0, 0. This will get us the top left of our image. The next two numbers, (100, 100) are the width and height of the rectangle.

Now that we have a rectangle set up, we can use the **Clone** method of Bitmap objects. Add this rather long line to your code:

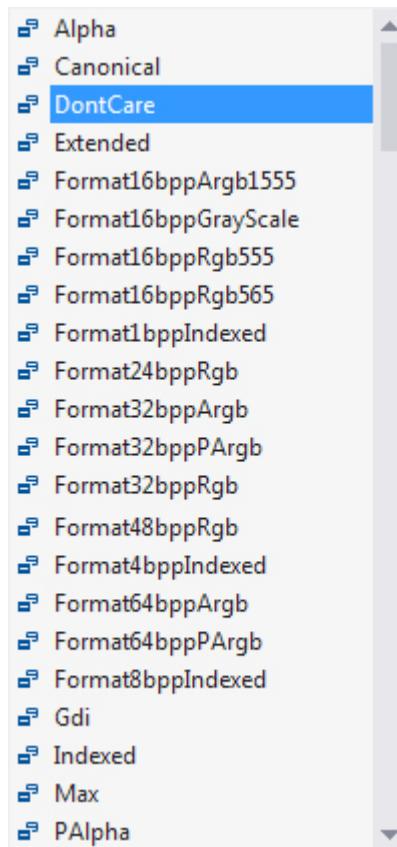
```
Bitmap cloneImage = bmp.Clone(rect, System.Drawing.Imaging.PixelFormat.DontCare);
```

The first thing we do is to set up a new Bitmap image called **cloneImage**. After an equal sign we have our previous Bitmap object (**bmp**). This, remember, holds the image from the PictureBox. After typing a dot, add the **Clone** method. The Clone method takes two arguments between its round brackets. The first is a rectangle object. This is needed in order to tell C# which area of the image you want to clone. For us, this was an area 100 by 100, starting at position 0, 0.

The second argument is rather curious. It's this:

System.Drawing.Imaging.PixelFormat.DontCare

C# needs to know the pixel format for the image. This refers to the quality of each pixel in the image you want cloned. **PixelFormat** is part of the **System.Drawing.Imaging** namespace. Type a dot after **PixelFormat** and you'll see the following list of options:



We're not too fussy about the image quality so we've gone for the splendidly-named **DontCare** option.

The only other thing to do is to transfer the cloned image to the PictureBox:

```
pictureBox1.Image = cloneImage;
```

The whole of your code should now look like this:

```
Bitmap bmp = new Bitmap(pictureBox1.Image);
Rectangle rect = new Rectangle(0, 0, 100, 100);
Bitmap cloneImage = bmp.Clone(rect, System.Drawing.Imaging.PixelFormat.DontCare);
pictureBox1.Image = cloneImage;
```

Try it out. Make sure there is an image loaded into your PictureBox (you can add the **if** statement that checks for this, if you like). Click your Clone button. Before the image is cloned ours looked like this:



After the clone, we're left with this:



Manipulating Image Pixels

You can manipulate each pixel of your image to get some really nice effects. We're going to add a colour filter to an image. A red filter added to the London Bridge image above, for example, would look like this:



The code for manipulating pixels, is however, slightly more difficult than for rotating, flipping and cloning. We'll need to get a pixel's colour, create a new colour based on that pixel's colour, then set a new pixel colour.

Add a new button to your form. Double click the button to open up its code stub. Create a Bitmap from your PictureBox as before:

```
Bitmap bmp = new Bitmap(pictureBox1.Image);
```

Now add these two variables to your code:

```
int x, y;
```

We need these because we're going to be using a double for loop, one inside the other. Add this double for loop to your code:

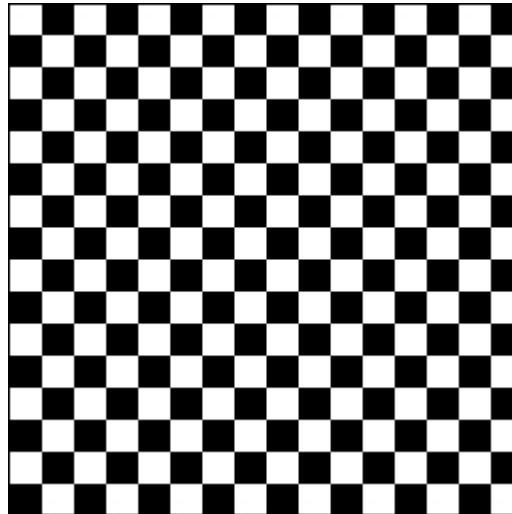
```
for (x = 0; x < bmp.Width; x++)
{
    for (y = 0; y < bmp.Height; y++)
    {
        }
}
```

The outer loop goes from 0 to the width of the Bitmap object called **bmp**. The inner loop goes from 0 to the height of the **bmp** object. Inside the inner loop, we'll get the **x** and **y** values. The first time round the outer loop the **x** variable will be 0. The inner loop is then executed in full, with the **y** variable being used. So you'd get these values the first time round:

```
x=0; y=1
x=0; y=2
x=0; y=3
x=0; y=4
x=0; y=5
x=0; y=6
```

etc,

If that's not too clear, take a look at this checkerboard pattern:



This pattern is a square 16 pixels by 16. What our two loops do is to go from left to right first of all. At the end of the line, we then drop down and go through the second row. All rows are then accessed from top to bottom.

Add this line to your inner for loop:

```
Color old_pixel_colour = bmp.GetPixel( x, y );
```

We're setting up a Color object here and calling it **old_pixel_colour**. After the equal sign we have this:

```
bmp.GetPixel( x, y );
```

We're accessing the Bitmap object called **bmp**, which holds the image from the PictureBox. After a dot we have an inbuilt method called **GetPixel**. This gets the colour of a pixel. In between round brackets you type the coordinates of the pixel whose colour you want to grab. We have **x, y** inside the round brackets. The first time round our outer loop **x** will have a value of 0. The first time round our inner loop, **y** will also have a value of 0. So the first pixel we grab is at 0, 0 from our image, which would be the first white square top, left of our checkerboard image. The second time round the inner loop the **x** and **y** values will be 0 and 1, which would get us the first black square from the left on the top row of the checkerboard image.

The next line to add to your inner loop is this:

```
Color new_pixel_colour = Color.FromArgb( old_pixel_colour.R, 0, 0 );
```

We're now setting up a new **Color** object and calling it **new_pixel_colour**. After an equal sign, we have this:

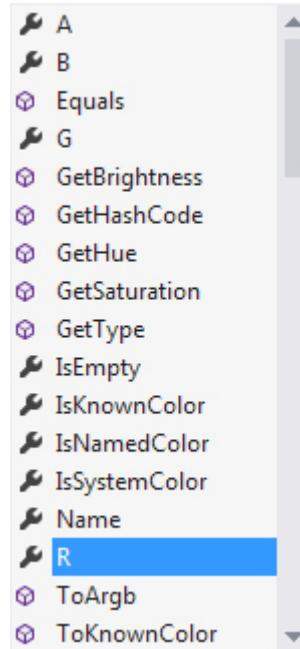
```
Color.FromArgb( old_pixel.colour.R, 0, 0 );
```

What we're doing here is creating a new colour based on the old one. To do that, we're using a method of the Color object called **FromArgb**. Ignore the **A** part for now (it stands for Alpha, and we'll get to it soon). The **rgb** part stands for Red, Green, Blue. This means the Red, Green, and Blue parts of a pixel. Our old pixel colour was white. This has a RGB of 255, 255, 255. A value of 255 means turn the colour full on. So the red part is turned full on, the green part is turned full on and the blue part is turned full on. This gives you a colour of white.

To turn the old white colour into red, we need to switch the red full on and the green and blue parts full off. This is done between the round brackets of **FromArgb**:

```
old_pixel.colour.R, 0, 0
```

When you type a dot after the variable **old_pixel.colour** you'll see the following list:



The B, G and R items allow you to get the Red, Blue and Green colour parts of a pixel. We're just getting the Red part. We're setting the Green and Blue parts to 0, meaning don't get these colours at all.

Remember, though, it's the just the Red part of the old colour. The old colour was white, which is RGB 255, 255, 255. We just want the first 255, the red part. We're setting the other 255's to 0.

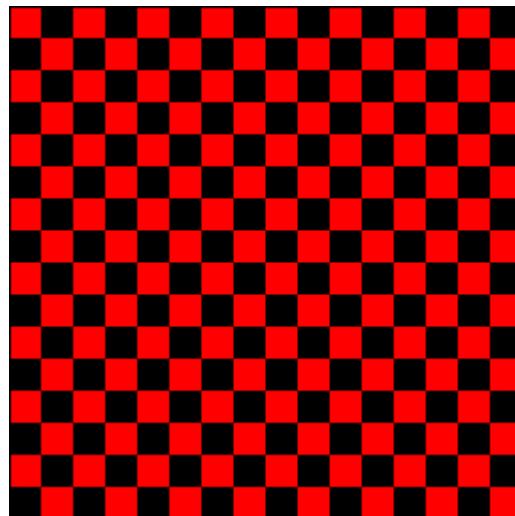
Now that we have a new colour from the old one, we can set a pixel. Add this line to your inner loop:

```
bmp.SetPixel(x, y, new_pixel.colour);
```

The **SetPixel** method sets a colour for a pixel. In between round brackets you first specify the location for your new pixel. Our location is held in the **x** and **y** variables, which will be 0, 0 the first time round both loops. After a comma, we type the new colour we want for the pixel in this position. So we've changed the pixel from white to red.

The second time round our inner for loop we get a black square. The colour black has a RGB value of 0, 0, 0. When we use **FromArgb** again we still get just the red part of those three numbers. The red part is 0. The new colour will, therefore, still be black.

When the loop is finished, the code above would turn a black and white checkerboard pattern to this:



If we wanted the checkerboard to go from white and black to yellow and black the **FromArgb** line would be this:

```
Color.FromArgb( old_pixel.colour.R, old_pixel.colour.G, 0 );
```

This time, we're grabbing the Red and Green parts of the old colour. If the old colour is white (255, 255, 255) then the new RGB is 255, 255, 0. Turning red full on, green full on and blue off gives you yellow.

The final line to add is outside of the two loops:

```
pictureBox1.Image = bmp;
```

This sets the PictureBox image to the new version of the **bmp** image.

Here's what your code should look like:

```

private void button4_Click(object sender, EventArgs e)
{
    Bitmap bmp = new Bitmap(pictureBox1.Image);

    int x, y;

    for (x = 0; x < bmp.Width; x++)
    {
        for (y = 0; y < bmp.Height; y++)
        {
            Color old_pixel_colour = bmp.GetPixel(x, y);

            Color new_pixel_colour = Color.FromArgb(old_pixel_colour.R, 0, 0);

            bmp.SetPixel(x, y, new_pixel_colour);
        }
    }

    pictureBox1.Image = bmp;
}

```

Run your programme and load an image into your PictureBox. Click your Filter button and it should turn red. As an exercise, amend your code so that the filter is blue instead of red.

Finally, a word about the **A** part of the **FromArgb** method.

The **A** stands for Alpha. An Alpha value is a measure of how transparent you want the colour of your pixels. A value of 0 means fully transparent, while a value of 255 means fully opaque. The default is 255. If you wanted to turn the opacity down a bit, you'd do it like this:

```
Color new_pixel_colour = Color.FromArgb( 100, old_pixel_colour.R, 0, 0);
```

The Alpha value goes as the first argument to **FromArgb**. Here's our London Bridge image again. The one on the left has an Alpha value of 255. The image on the right has an Alpha value of 100 set for it:



Invert Pixel Colours

Colour inversion is the colour left over when you deduct one RGB value from 255, which is the highest RGB value you can have. For example, take white. It has an RGB of 255, 255, 255. The invert is 0, 0, 0. This is because we first take 255 and deduct the Red part, which also 255. 255 minus 255 is obviously 0. Do the same for the Green and Blue values and you end up with black. Another example is this colour: RGB(255, 182, 193). This is a light pink. To get the inverted colour we first deduct the red part from 255, which leaves 0. To get the green part we deduct 182 from 255 which is 73. To get the blue part we deduct 193 from 255, which is 62. So the new inverted colour is RGB(0, 73, 62), which is a sort of green/blue colour.

The code to do an inversion on a whole image is just about the same as for changing the pixel colours above. The only difference is the **FromArgb** line:

```
Color new_pixel_colour = Color.FromArgb(100, old_pixel_colour.R, 0, 0);
```

We need to deduct the old pixels value from 255. The new line, then is this:

```
Color new_pixel_colour = Color.FromArgb( 255 - old_pixel_colour.R,
255 - old_pixel_colour.G, 255 - old_pixel_colour.B);
```

The whole of the code to invert a colour is as follows:

```
Bitmap bmp = new Bitmap(pictureBox1.Image);

int x, y;

for (x = 0; x < bmp.Width; x++)
{
    for (y = 0; y < bmp.Height; y++)
    {
        Color old_pixel_colour = bmp.GetPixel(x, y);
        Color new_pixel_colour = Color.FromArgb(255 - old_pixel_colour.R,
                                              255 - old_pixel_colour.G,
                                              255 - old_pixel_colour.B);

        bmp.SetPixel(x, y, new_pixel_colour);
    }
}

pictureBox1.Image = bmp;
```

As you can see, it's the same as before except for that one **new_pixel_colour** variable line.

Add a button to your form. Change the text property to **Invert**. Add the code above. When you run your form, here's what an image would look like before the button is clicked:



And here's what it looks like after it has been inverted:



Saving an Image

You'll be glad to know that saving an image is quite easy!

Add a **SaveFileDialog** control to your form from the C# toolbox. Change the **Name** property to **saveFD**. Add a new button to your form and change the **Text** property to **Save**. Double click your button to get at the code stub. Now add the following:

```
string save_path = "";
saveFD.InitialDirectory =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

saveFD.FileName = "default";
saveFD.Filter = "JPEG|*.jpeg";
```

We've already covered Save dialogue boxes in a previous section, so we won't explain what it all means here. But we're just setting the save directory to default to the **My Documents** folder in Windows. The filter has been set to JPEG images. You'll see why in a moment.

Now add the following if statement:

```
if (saveFD.ShowDialog() != DialogResult.Cancel)
{
}
```

As the first line of your if statement add these two lines:

```
save_path = saveFD.FileName;

Bitmap bmp = new Bitmap(pictureBox1.Image);
```

The dialogue boxes, remember, just return a file path – they don't actually save anything. We're putting the **FileName** into the string variable we've called **save_path**. The next line just gets the picture from the PictureBox, as before.

To do the actual saving, you can use the **Save** method of Bitmap objects.

The Save method has many alternatives. But one of them uses two arguments: a file path and an image format. Add this line to your if statement:

```
bmp.Save( save_path, System.Drawing.Imaging.ImageFormat.Jpeg );
```

In between the round brackets we first have our file path. After a comma, we have this:

```
System.Drawing.Imaging.ImageFormat.Jpeg
```

ImageFormat is a class in the **System.Drawing.Imaging** namespace. When you type the dot after **ImageFormat** you'll see a list appear. This one:



This is a list of image formats that the Bitmap can save to. We've gone for the JPEG image format.

As the final line of the Save code you can add a message box, if you like, The whole of the code would then look like this:

```
string save_path = "";
saveFD.InitialDirectory = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

saveFD.FileName = "default";
saveFD.Filter = "JPEG|*.jpeg";

if (saveFD.ShowDialog() != DialogResult.Cancel)
{
    save_path = saveFD.FileName;

    Bitmap bmp = new Bitmap(pictureBox1.Image);

    bmp.Save(save_path, System.Drawing.Imaging.ImageFormat.Jpeg);

    MessageBox.Show("Image Saved");
}
```

Run your programme and test it out. Load an image into your PictureBox. Click one of your buttons, such as Filter or Invert. Then click your Save button. You should find that you'll have a new image in your **My Documents** folder.

And that's it for image manipulation. We hope it has whetted your appetite to explore more! And that's it, too, for this beginner's book on C# .NET. We hope you enjoyed it, and wish you all the best for your future programming career. Good luck!

Answers To Exercises

Exercise A

```
int answerSubtract;  
int numberOne = 12;  
int numberTwo = 4;  
  
answerSubtract = 50 - numberOne - numberTwo;  
  
MessageBox.Show(answerSubtract.ToString());
```

[< Back to Exercise A](#)

Exercise B

Your first answer should be 150. Because the variables were set up as **int**, however, your second answer should be 100. Try changing the int variables to float and you'll get an answer of 150. This is the answer you would expect because 75 divided by 50 is 1.5. 1.5 multiplied by 100 is 150. If you set your variables up as int then the .5 of 1.5 gets chopped off, leaving just 1 to multiplied by 100.

[< Back to Exercise B](#)

Exercise C

```
private void btnPoint_Click(object sender, EventArgs e) {  
  
    txtDisplay.Text = txtDisplay.Text + btnPoint.Text;  
  
}
```

[< Back to Exercise C](#)

Exercise D

```
public Form1() {  
  
    double total1 = 0;  
    double total2 = 0;  
  
    bool plusButtonClicked = false;  
    bool minusButtonClicked = false;  
    bool divideButtonClicked = false;  
    bool multiplyButtonClicked = false;  
  
    //=====  
    //      NUMBER BUTTONS  
    //=====  
    private void btnOne_Click(object sender, EventArgs e) {  
        txtDisplay.Text = txtDisplay.Text + btnOne.Text;  
    }  
  
    private void btnTwo_Click(object sender, EventArgs e) {  
        txtDisplay.Text = txtDisplay.Text + btnTwo.Text;  
    }  
  
    private void btnThree_Click(object sender, EventArgs e) {  
        txtDisplay.Text = txtDisplay.Text + btnThree.Text;  
    }  
  
    //OTHER NUMBER BUTTONS HERE  
  
    private void btnPoint_Click(object sender, EventArgs e) {  
        txtDisplay.Text = txtDisplay.Text + btnPoint.Text;  
    }  
  
    //=====  
    //      MINUS, PLUS, DIVIDE, MULTIPLY BUTTONS  
    //=====  
    private void btnMinus_Click(object sender, EventArgs e) {  
        total1 = total1 + double.Parse(txtDisplay.Text);  
        txtDisplay.Clear();  
  
        minusButtonClicked = true;  
        plusButtonClicked = false;  
        divideButtonClicked = false;  
        multiplyButtonClicked = false;  
    }  
  
    private void btnPlus_Click(object sender, EventArgs e) {  
        total1 = total1 + double.Parse(txtDisplay.Text);  
        txtDisplay.Clear();  
    }  
}
```

```

        plusButtonClicked = true;
        minusButtonClicked = false;
        divideButtonClicked = false;
        multiplyButtonClicked = false;
    }

private void btnMultiply_Click(object sender, EventArgs e) {
    total1 = total1 + double.Parse(txtDisplay.Text);
    txtDisplay.Clear();

    minusButtonClicked = false;
    plusButtonClicked = false;
    divideButtonClicked = false;
    multiplyButtonClicked = true;
}

private void btnDivide_Click(object sender, EventArgs e) {
    total1 = total1 + double.Parse(txtDisplay.Text);
    txtDisplay.Clear();

    minusButtonClicked = false;
    plusButtonClicked = false;
    divideButtonClicked = true;
    multiplyButtonClicked = false;
}

//=====
//      EQUALS BUTTON
//=====

private void btnEquals_Click(object sender, EventArgs e) {

    if (plusButtonClicked == true) {
        total2 = total1 + double.Parse(txtDisplay.Text);
    }
    else if (minusButtonClicked == true) {
        total2 = total1 - double.Parse(txtDisplay.Text);
    }
    else if (multiplyButtonClicked == true) {
        total2 = total1 * double.Parse(txtDisplay.Text);
    }
    else if (divideButtonClicked == true) {
        total2 = total1 / double.Parse(txtDisplay.Text);
    }

    txtDisplay.Text = total2.ToString();
    total1 = 0;
}

```

```

private void btnClear_Click(object sender, EventArgs e) {
    plusButtonClicked = false;
    minusButtonClicked = false;
    divideButtonClicked = true;
    multiplyButtonClicked = false;

    txtDisplay.Clear();
}

}

```

[< Back to Exercise D](#)

Exercise E

```

double total1 = 0;
string theOperator;

//=====
//      NUMBER BUTTONS ARE THE SAME AS BEFORE
//=====

//=====
//      THE PLUS, MINUS, TIMES AND DIVIDE  BUTTONS
//=====

private void btnPlus_Click(object sender, EventArgs e) {
    total1 = total1 + double.Parse(txtDisplay.Text);
    theOperator = "+";
    txtDisplay.Clear();
}

private void btnMinus_Click(object sender, EventArgs e) {
    total1 = total1 + double.Parse(txtDisplay.Text);
    theOperator = "-";
    txtDisplay.Clear();
}

private void btnTimes_Click(object sender, EventArgs e) {
    total1 = total1 + double.Parse(txtDisplay.Text);
    theOperator = "*";
    txtDisplay.Clear();
}

private void btnDivide_Click(object sender, EventArgs e) {
    total1 = total1 + double.Parse(txtDisplay.Text);
    theOperator = "/";
    txtDisplay.Clear();
}

```

```

//=====
//      THE EQUALS BUTTON.
//=====

private void btnEquals_Click(object sender, EventArgs e) {

    double num2;
    double answer;

    num2 = double.Parse(txtDisplay.Text);

    switch (theOperator) {
        case "+":
            answer = total1 + num2;
            txtDisplay.Text = answer.ToString();
            total1 = 0;
            break;
        case "-":
            answer = total1 - num2;
            txtDisplay.Text = answer.ToString();
            total1 = 0;
            break;
        case "*":
            answer = total1 * num2;
            txtDisplay.Text = answer.ToString();
            total1 = 0;
            break;
        case "/":
            answer = total1 / num2;
            txtDisplay.Text = answer.ToString();
            total1 = 0;
            break;
        default:
            answer = 0;
            break;
    }

}

```

[< Back to Exercise E](#)

Exercise F

```

int age;

age = int.Parse(textBox1.Text);

if (age < 17) {
    MessageBox.Show("Still a youngster.");
}

```

```

}
else if (age >= 16 && age <= 24) {
    MessageBox.Show("Fame beckons!");
}

else if (age >= 25 && age < 40 ) {
    MessageBox.Show("There's still time.");
}
else {
    MessageBox.Show("Oh dear, you've probably missed it!");
}

```

[< Back to Exercise F](#)

Exercise G

Extra points if you spotted that that loopEnd could be $i < (\text{loopEnd} + 1)$

[< Back to Exercise G](#)

Exercise H

```

int answer = 0;
int loopStart;
int loopEnd;

loopStart = int.Parse(textBox1.Text);
loopEnd = int.Parse(textBox2.Text);
int multiplyBy = int.Parse(textBox3.Text);

listBox1.Items.Clear();

for ( int i = loopStart; i <= loopEnd; i++) {
    answer = multiplyBy * i;
    listBox1.Items.Add( i + " times " + multiplyBy + " = " +
        answer.ToString());
}

```

[< Back to Exercise H](#)

Exercise I

The letter count never gets beyond zero because of this line:

```
letter = strText.Substring(0, 1);
```

Substring(0, 1) means start at the first character in the string (character 0) and grab 1 character. You need to use the loop variable to move through the text:

```
letter = strText.Substring(i, 1);
```

[< Back to Exercise I](#)

Exercise J

```
//=====
//      MULTIPLY BUTTON:
//=====

int number1;
int number2;
int returnValue = 0;

number1 = int.Parse(textBox1.Text);
number2 = int.Parse(textBox2.Text);

returnValue = Multiply(number1, number2);

MessageBox.Show(returnValue.ToString());


//=====
//      MULTIPLY METHOD
//=====

private int Multiply(int firstNumber, int secondNumber) {

    int answer;

    answer = firstNumber * secondNumber;

    return answer;
}

//=====
//      DIVIDE BUTTON
//=====

int number1;
int number2;
int returnValue = 0;

number1 = int.Parse(textBox1.Text);
number2 = int.Parse(textBox2.Text);

returnValue = Divide(number1, number2);

MessageBox.Show(returnValue.ToString());
```

```

//=====
//      DIVIDE METHOD
//=====

private int Divide(int firstNumber, int secondNumber) {
    int answer;

    answer = firstNumber / secondNumber;

    return answer;
}

```

[< Back to Exercise J](#)

Exercise K

array[0] doesn't get used because the for loop is starting at 1.

[< Back to Exercise K](#)

Exercise L

```

for (int i = 0; i != (aryTimes.Length); i++) {
    storeAnswer = (i + 1) * times;
    aryTimes[i] = storeAnswer;
    listBox1.Items.Add(times + " times " + (i + 1) + " = " + aryTimes[i]);
}

```

[< Back to Exercise L](#)

Exercise M

```

string emailAddress = "enquiry@me.co.uk";
string result = "";

result = emailAddress.Substring(11, 5);

if (result == "co.uk") {
    MessageBox.Show("Email Address OK " + result);
}
else {
    MessageBox.Show("Bad Email Address " + result);
}

```

[< Back to Exercise M](#)

Exercise N

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e) {  
  
    if (listBox1.SelectedIndex == 1) {  
        loadListBox();  
    }  
  
    if (listBox1.SelectedIndex == 0) {  
        loadListBoxCheques();  
    }  
}  
  
private void loadListBox() {  
  
    listBox2.Items.Clear();  
  
    listBox2.Items.Add("American Express");  
    listBox2.Items.Add("Discover");  
    listBox2.Items.Add("Mastercard");  
    listBox2.Items.Add("Visa");  
}  
  
private void loadListBoxCheques() {  
  
    listBox2.Items.Clear();  
  
    listBox2.Items.Add("Business Cheque");  
    listBox2.Items.Add("eCheque");  
    listBox2.Items.Add("Personal Cheque");  
}
```

[< Back to Exercise N](#)

Exercise O

```
class HappyBirthday {  
  
    //=====  
    // CLASS VARIABLES  
    //=====  
    private int numberOfPresents;  
    private string birthdayMessage;  
    private bool birthdayParty;
```

```

//=====
// DEFAULT CONSTRUCTOR
//=====
public HappyBirthday()
{
    numberOfPresents = 0;
    birthdayParty = false;
}

//=====
// METHOD
//=====
private string getMessage(string givenName)
{
    string theMessage;

    theMessage = "Happy Birthday " + givenName + "\n";
    theMessage += "Number of presents = ";
    theMessage += numberOfPresents.ToString() + "\n";

    if (birthdayParty == true)
    {
        theMessage += "Hope you enjoy the party!";
    }
    else
    {
        theMessage += "No party = sorry!";
    }

    return theMessage;
}

//=====
// READ AND WRITE PROPERTY
//=====
public string MyProperty
{
    get { return birthdayMessage; }

    set { birthdayMessage = getMessage(value); }
}

//=====
// WRITE-ONLY PROPERTY
//=====
public int PresentCount
{
    set { numberOfPresents = value; }
}

```

```

    }

    public bool hasParty
    {
        set { birthdayParty = value; }
    }

}

```

[< Back to Exercise O](#)

Exercise P

```

private void labelUpdate() {

    label5.Text = "Record " + (inc + 1) + " of " + ds1.Tables["Workers"].Rows.Count;
}

```

Call this method from the NavigateRecords method and the Form Load event.

[< Back to Exercise P](#)

Exercise Q

```

private void btnFind_Click(object sender, EventArgs e) {

    string searchFor = "";
    string searchOn = "";
    string searchString = "";
    int results = 0;

    if (txtFind.Text.Trim() == "") {
        MessageBox.Show("Nothing to Search For");
        return;
    }

    if (cb1.Text == "Last Name") {
        searchFor = txtFind.Text.Trim();
        searchOn = "last_Name=";
        searchString = searchOn + "" + searchFor + "";
    }

    if (cb1.Text == "Job Title") {
        searchFor = txtFind.Text.Trim();
        searchOn = "job_Title=";
        searchString = searchOn + "" + searchFor + "";
    }
}

```

```
DataRow[] returnedRows;

returnedRows = ds1.Tables["Workers"].Select(searchString);

results = returnedRows.Length;

if (results > 0) {
    string searchMessage = "";

    DataRow dr1;

    dr1 = returnedRows[0];

    searchMessage = dr1["first_Name"].ToString() + " " +
dr1["last_Name"].ToString();

    searchMessage += ", Job Title: " + dr1["job_Title"].ToString();

    MessageBox.Show(searchMessage);
}
else {
    MessageBox.Show("No such Record");
}

}
```

[< Back to Exercise Q](#)